AIX Version 3 for
RISC System/6000™

Calls and Subroutines Reference: Kernel

Volume 5

SC23-2198-00

First Edition (March 1990)

This edition of the *AIX Calls and Subroutines Reference for IBM RISC System/6000*, Volume 5, *Kernel Reference*, applies to Version 3.0 of the AIX IBM Base Operating System and to all subsequent releases of this product until otherwise indicated in new releases or technical newsletters.

Example device driver and device method source code can be found in the **/usr/lpp/bos/samples** directory once the Base Development Libraries and Include Files component of the Base Application Development Toolkit licensed program has been installed. These source code examples are only intended to assist in the development of a working sortware program. These examples do not function as written: ADDITIONAL CODE IS REQUIRED. In addition, the source code examples may not compile and/or bind successfully as written.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758–3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks and Acknowledgements

About This Book

AIX Calls and Subroutines Reference for IBM RISC System/6000, SC23–2198, is divided into the following four major sections:

- Volumes 1 and 2, *Calls and Subroutines Reference*: *Base Operating System*, contains reference information about the system calls, subroutines, functions, macros, and statements associated with AIX base operating system runtime services, communications services, and device services.
- Volumes 3 and 4, *Calls and Subroutines Reference*: *User Interface*, contain reference information about the AIXwindows widget classes, subroutines, and resource sets; the AIXwindows Desktop resource sets; the Enhanced X–Windows subroutines, macros, protocols, extensions, and events; the X–Window toolkit subroutines and macros; and the curses and extended curses subroutine libraries.
- Volume 5, *Calls and Subroutines Reference*: *Kernel Reference*, contains reference information about kernel services, device driver operations, file system operations, subroutines, the configuration subsystem, the communications subsystem, the high function terminal (HFT) subsystem, the logical volume subsystem, the printer subsystem, and the SCSI subsystem.
- Volume 6, *Calls and Subroutines Reference*: *Graphics*, contains reference information and example programs for the Graphics Library (GL) and the AIXwindows Graphics Support Library (XGSL) subroutines.

This volume, *Calls and Subroutines Reference*: *Kernel Reference,* is a technical reference detailing all system services available for writing kernel extensions. In particular, this reference describes existing kernel services and the interfaces needed for programming kernel extensions. Possible types of kernel extensions include device drivers, system calls, kernel services or virtual file systems.

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

Who Should Use This Book

This book is intended for systems programmers wishing to extend the AIX kernel. Readers should be familiar with operating system concepts and kernel programming. Those wishing a review of this background should see *Kernel Extensions and Device Support Programming Concepts* for an overview.

How to Use This Book

Overview of Contents

The *Kernel Reference* contains two parts. Part 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. Extended descriptions of device driver routines and related data structures are discussed here.
- Interface requirements for writing virtual file systems. Extended descriptions of virtual file system routines are provided.

Part 2 details the interface requirements for AIX subsystem programming. This information describes individual device drivers and the use of the device-related subroutines (**open, close, read, write, ioctl**) that control them. The AIX subsystems include:

- The communications I/O subsystem. This chapter contains information about features common to all communications device drivers, as well as details about specific adapters. These include the Ethernet, Token-Ring, X.25, and MPQP adapters.
- The configuration subsystem. This chapter includes a description of the configuration databases, requirements for writing configuration methods, and a description of existing configuration routines.
- The high function terminal (HFT) subsystem. This chapter describes the use of subroutines and structures needed to control the high function terminal.
- The logical volume manager subsystem. This chapter describes the logical volume device driver and how it accesses the underlying physical devices.
- The printer addition management subsystem. This chapter describes routines needed for adding a new type of printer to the system.
- The SCSI subsystem. This chapter describes the SCSI tape, disk, and CD-ROM device drivers.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.
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.

Related Publications

The following books contain information about or related to device drivers and other kernel extensions.

- *AIX Files Reference for IBM RISC System/6000*, Order Number SC23–2200.
- *AIX General Programming Concepts for IBM RISC System/6000*, Order Number SC23–2205.
- *AIX Kernel Extensions and Device Support Programming Concepts for IBM RISC System/6000*, Order Number SC23–2207.
- *IBM RISC System/6000 Hardware Technical Reference — 7012 POWERstation and POWERserver*, Order Number SA23–2660.
- *IBM RISC System/6000 Hardware Technical Reference — 7013 and 7016 POWERstation and POWERserver*, Order Number SA23–2644.
- *IBM RISC System/6000 Hardware Technical Reference — 7015 POWERserver*, Order Number SA23–2645.
- *IBM RISC System/6000 POWERstation and POWERserver Hardware Technical Reference — General Information*, Order Number SA23–2643.
- *IBM RISC System/6000 POWERstation and POWERserver Hardware Technical Reference — Options and Devices*, Order Number SA23–2646.
- *IBM RISC System/6000 Problem Solving Guide*, Order Number SC23–2204.

Ordering Additional Copies of This Book

To order additional copies of this book, use Order Number SC23–2198–00.

Table of Contents

Part 1. Programming in the Kernel Environment

Chapter 1. Kernel Services

ackque Kernel Service

Purpose

Sends an acknowledge device queue element.

Syntax

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

int ackque (*qe*, *flags*, *results***)**
struct ack_qe ****qe*;**
int *flags***;**
int *results***;**

Parameters

qe	Specifies the address of the acknowledgment queue element.
flags	Specifies the operation options.
results	Specifies the operation results for a synchronous request or an interrupt on error request.

Description

The **ackque** 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 once before the loading of any kernel extensions referencing these services.

The **ackque** service is called by a device queue server (typically a kernel process) to send an acknowledgment. The operation option flags and the path type control the sending of an acknowledgment. Depending on the type of acknowledgment requested, different amounts of status information are returned.

The acknowledgment is only sent if both the path type and the operation options indicate that an acknowledgment is to be sent. The **deque** kernel service has more detailed information.

There are two types of acknowledgments: *solicited* acknowledgment and *unsolicited* acknowledgment. A solicited acknowledgment is sent in response to a request that was dequeued with the suppress option. All other acknowledgments are considered unsolicited.

If the suppress option is used with the **deque** service, the device queue's server is responsible for explicitly generating the acknowledgment by calling the **ackque** server. The original request queue element is unavailable in this case. This is overcome by the server remembering the operation options and passing them as the *flags* parameter.

A path to a device queue may be destroyed before the active queue element is totally processed. If this happens, no acknowledgment is generated when the **ackque** service is called. Instead, the queue element is discarded with no error reported.

Use of Virtual Interrupt Handlers

For compatibility purposes, when an acknowledgment is sent through a path that was set up with an acknowledgment type of interrupt (INTR_ACK), a registered virtual interrupt handler is called. The **ackque** service determines which virtual interrupt handler to call by determining the sublevel associated with the acknowledge queue element. If the **qe–>data[5]** field in the acknowledgment queue element is positive (that is, the most significant bit is a 0), then the sublevel specified when the path was created is used. Otherwise, the value in the field is used as the sublevel for calling the correct virtual interrupt handler.

ACKNOWLEDGE	TYPE VALUE	PARAMETER ONE	PARAMETER TWO
None	NO_ACK	n/a	n/a
Short	SHORT_ACK	Event mask	n/a
Long	LONG_ACK	Acknowledge device queue identifier	Queue element priority
Interrupt	INTR_ACK	n/a	Interrupt level and sublevel

Virtual interrupt handlers are registered by using the **vec_init** service. The virtual interrupt handler is directly called by the **ackque** service and executes in the process environment of the caller.

Execution Environment

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

Return Value

RC_GOOD Indicates successful completion.

No error is reported if the queue element is discarded.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **deque** kernel service, **vec_init** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

add_arp_iftype Kernel Service

Purpose

Adds an interface type to the Network ARP Switch Table Interface (NASTI).

Syntax

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

int add_arp_iftype(*if_type*, *af*, *ioctl*, *resolve*, *whohas*, *arptfree***)**

u_short	*if_type, af*;
int	(**ioctl*)();
int	(**resolve*)();
int	(**whohas*)();
int	(**arptfree*)();

Parameters

if_type	Uniquely identifies the type of a network interface (for example, Ethernet or token ring). Interface types are defined in the **/usr/include/sys/devinfo.h** file.
af	Specifies the address family that the specified ARP routines are able to handle.
ioctl	Specifies the ARP **ioctl** handler.
resolve	Specifies the ARP resolve handler.
whohas	Specifies a function for transmitting ARP request packets.
arptfree	Specifies a function that frees ARP entries and reclaims resources.

Description

The **add_arp_iftype** kernel service adds an interface type to the Network ARP Switch Table Interface (NASTI).

Example

The **add_arp_iftype** kernel service is invoked by:

```
add_arp_iftype(DD_EN, AF_INET, arpioctl, arpresolve);
```

Return Values

0	Indicates a successful operation.
EEXIST	Indicates that the type specified by the *if_type* parameter for the specified address *af* had already been added to the table.
ENOSPC	Indicates that no free slots were left in NASTI.
EINVAL	Indicates an error in the input parameters.

Execution Environment

The **add_arp_iftype** kernel service can be called from either the process or interrupt environment. The functions specified by the *ioctl*, *resolve*, *whohas*, and *arptfree* parameters are can also be called in either the process or interrupt environments.

Implementation Specifics

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

Related Information

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

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*, *af_netmatch*, *af_hash***)**
**struct domain ****domain*;
int (**af_netmatch***);**
int (**af_hash***);**

Parameters

domain	Specifies the domain of the address family.
af_netmatch	Specifies a function that the generic routing code calls to determine if two addresses are on the same network. The function should be of the form: **int** *af_netmatch* (s1, s2) **struct sockaddr** s1; **struct sockaddr** s2; The *af_netmatch* parameter should return 1 if the two addresses are on the same network. Otherwise, it should return a 0 (zero).
af_hash	Specifies a function that the generic routing code calls to determine routing hash values. The function should be of the form: *af_hash* (sa, hp) **struct sockaddr** *sa **struct afhash** *hp;

Description

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

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.

Execution Environment

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

Example

1. 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, inet_netmatch, inet_hash);
   ```

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

Implementation Specifics

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

Related Information

The **del_domain_af** kernel service.

Network Kernel Services in *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 a value of NET_OFF_LEVEL, the input handler specified by the *isr* parameter is called directly. Setting the *service_level* parameter to a value of NET_KPROC causes a network dispatch process to be scheduled. This dispatch process calls the subroutine identified by the *isr* parameter.
isr	Identifies the routine that is to serve 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. This parameter must be specified if the processing level specified by the *service_level* parameter is a value of NET_KPROC. Specifying NULL for this parameter generates a call to the input handler specified by the *isr* parameter, as in the following: (**isr*)(CommonPortion,Buffer); In this example, `CommonPortion` points to the network common portion (struct **arpcom**) of a network interface and `Buffer` is a pointer to a buffer (**mbuf**) containing an input packet.
af	Specifies the address family of the calling protocol. The *af* parameter must be specified if the *ifq* parameter is not NULL.

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 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

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

Related Information

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

Network Kernel Services in *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 (zero) and less than a value of 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 a value of NET_KPROC, a network interrupt scheduler calls the function specified by the *isr* parameter. If you set the *service_level* parameter to a value of NET_OFF_LEVEL, the **add_netisr** 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 an invalid service level.

Implementation Specifics

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

Related Information

The **del_netisr** kernel service.

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

This macro is part of AIX Base Operating System (BOS) Runtime.

Related Information

The **del_netopt** macro.

The **no** command.

Network Kernel Services in *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/addspace.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. This pointer can be obtained by using the **getadsp** kernel service.
vmhandle	Describes the virtual memory object that is being made addressable within a region of the specified address space.
offset	Specifies the offset in the virtual memory object and region that is being mapped. On the RISC System/6000, 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

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

Related Information

The **getadsp** kernel service, **as_det** kernel service.

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

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/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
#include <sys/addspace.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, this pointer can be obtained with the **getadsp** kernel service.
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.

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 the RISC System/6000, 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

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

Related Information

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

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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 that is being attached to.

Description

Each device queue can have an **attach** routine. This routine is optional and must be specified when the device queue is defined with the **creatd** kernel service. 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 of the 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

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

attchq Kernel Service

Purpose

Creates a path to a device queue.

Syntax

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

int attchq (*from_id, to_id, path_id, ptr***)**
cba_id *from_id*;
cba_id *to_id*;
cba_id **path_id*;
struct attchq **ptr*;

Parameters

from_id	Specifies the identifier of the requestor.
to_id	Specifies the identifier of the server.
path_id	Specifies the address of the returned path identifier.
ptr	Specifies the address of the acknowledge parameter structure.

Description

The **attchq** 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 once before the loading of any kernel extensions referencing these services.

The **attchq** service establishes how a requestor and a server communicate with each other. For a discussion of the device queue requestor and server model, see Understanding Device Queues. The *from_id* and *to_id* parameters give the identifiers of the requestor and the server of the device queue, respectively. These identifiers can be a queue identifier, a device identifier, or a process identifier. Neither identifier needs to be associated with the caller of the **attchq** service.

If a process identifier is specified, a path is established to the oldest device queue served by the process. If a device identifier is specified, a path is established to the device queue associated with the device identifier. If a queue identifier is used, then a path is established to that queue.

The server's **attach-device** routine is called if an **attach-device** routine is associated with the device queue and the *to_id* parameter is a device identifier.

The Acknowledgment Parameter Structure

The acknowledgment parameter structure consists of four fields: acknowledge-type, acknowledge depth-counter, and two other parameters. The latter two (Parameter One and Parameter Two) contain data whose meaning depends on the acknowledge type.

The Acknowledgment Type and How It Is Used

The acknowledge-type field specifies acknowledgment information returned when the processing of a queue element is completed. The four type options are:

NO_ACK	No acknowledgment is to be sent. Parameters One and Two have no meaning.
SHORT_ACK	Completion is to be acknowledged by posting an event. A short acknowledgement notifies the requestor by sending it an event notification using the **e_post** kernel service. Parameter One contains an event mask to be used as the *events* parameter for the **attchq** service. Parameter Two has no meaning.
LONG_ACK	Completion is to be acknowledged by sending a queue element. A long acknowledgment notifies the requestor by sending the requestor a queue element. Parameter One contains an acknowledge device queue identifier specifying the device queue to which the acknowledgment queue element is to be sent. If this identifier is NULL_CBA, an acknowledgement is sent to the first device queue associated with the *from_id* parameter. Parameter Two contains the queue element priority, which is a number from QE_BEST_PRTY to QE_WORST_PRTY. This priority is described in more detail with the **enque** service.
INTR_ACK	Completion is to be acknowledged by sending a virtual interrupt. A virtual interrupt acknowledgment notifies the requestor by calling its registered virtual interrupt handler with the acknowledge queue element. The requestor can use the **vec_init** service to define a virtual interrupt handler to receive the virtual interrupt queue element. For this acknowledge type, the virtual interrupt level and sublevel occupy the last 16 bits of Parameter Two. Of these 16 bits, the first 8 (high-order byte) are the virtual interrupt level (0 to 7) and the next 8 bits (low-order byte) are the virtual interrupt sublevel (0 to 255). The virtual interrupt level is ignored. Virtual interrupts should be used for compatibility purposes only. Parameter One has no meaning.

The Acknowledgment Depth Counter

Another part of the acknowledgment parameter structure is the acknowledgment depth counter. This counter places a limit on the number of acknowledgment queue elements that can be outstanding at any given time. Use of this counter prevents runaway consumption of queue elements in error situations. If the count is exceeded, the acknowledgment overrun count is increased. If zero is specified for the counter, it defaults to a value of one. The largest valid acknowledgment depth count is MAX_ACK_DEPTH.

Note: The kernel may or may not enforce the restriction on the size of the acknowledgment depth count.

In addition to the return code, the path identifier is also returned in the memory indicated by the *path_id* parameter. The path identifier is used by other device queue management services such as the **enque** kernel service.

Execution Environment

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

Return Values

RC_GOOD	Indicates a successful operation.
RC_NONE	Indicates that resources were unavailable. The path was not created.
RC_MAX	Indicates that the maximum number of paths was exceeded. The path was not created.

All other error values represent errors detected by the server's **attach-device** routine.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **enque** kernel service, **vec_init** kernel service, **e_post** kernel service.

The **attach–device** queue management routine.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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 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 that are passed by reference.

After the record buffer is complete and if auditing is enabled, the information is written by the **audit_svcfinis** kernel service.

Execution Environment

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

Return Values

0	Indicates a successful operation.
ENOSPC	Indicates that the kernel service is unable to allocate space for the new buffer.
EINVAL	Indicates that no valid audit record buffer exists.

Implementation Specifics

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

Related Information

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

Security Kernel Services in *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 ()
int;

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 SVC handler exit routine writes the records. This exit routine calls the **audit_svcfinis** kernel service to complete the records.

The result code is computed from the current **errno** value.

Execution Environment

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

Return Value

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

Implementation Specifics

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

Related Information

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

Security Kernel Services in *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. 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 that the **audit_svcstart** kernel service is called, this parameter is set to the actual table index. The system call should not reset it. It should be declared as a static.
numargs	Specifies the number of parameters to be included in the buffer for this record. These parameters are normally 0 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.

Example

1. You can invoke the **audit_svcstart** service with the following:

```
svcfoobar(int x, int y, int z)
 {
static int eventnum;
if (audit_svcstart("fubared", &eventnum, 2, x, y)) {
  audit_svcfinis();
  }
   ...
   body of svcfoobar
   ...
}
```

This allocates an audit event record buffer for the event **fubared** and copies the first and second arguments into it. The third argument is unnecessary and is not copied.

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.

Implementation Specifics

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

Related Information

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

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

bawrite Kernel Service

Purpose

Writes the specified buffer's 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 for the buffer to be written.

Description

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

The article entitled Using the Buffer Cache write Services briefly describes how the three buffer cache write routines work.

Execution Environment

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

Return Values

0	Indicates successful completion.
Errno global variable	Indicates that an I/O error has occurred.

Implementation Specifics

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

Related Information

The **bwrite** kernel service.

Block I/O Buffer Cache Services: Overview, I/O Kernel Services in *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.

Using the Buffer Cache write Services briefly describes how the three buffer cache write routines work.

Execution Environment

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

Implementation Specifics

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

Related Information

The **brelse** kernel service.

Block I/O Buffer Cache Services: Overview, I/O Kernel Services in *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 marks as busy or writing any dirty buffer whose block is on the specified device. When 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

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

Related Information

The **bwrite** kernel service.

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

binval Kernel Service

Purpose

Invalidates all of the specified device's blocks in the buffer cache.

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 invalidates, or makes nonreclaimable, all of the specified device's blocks in the buffer cache. Before removing the device from the system, the **binval** service should be called to remove all of a device's blocks from the buffer cache.

All of the device's blocks should have been flushed before calling the **binval** service. Typically, these blocks are flushed after the last close of the device.

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

Execution Environment

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

Implementation Specifics

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

Related Information

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

Block I/O Buffer Cache Services: Overview, I/O Kernel Services in *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, then 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

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

Related Information

The **bwrite** kernel service.

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

bread Kernel Service

Purpose

Reads the specified block's 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's buffer header is returned. Otherwise, a free buffer is assigned to the specified block and the block's data is read into the buffer. The **bread** service waits for I/O to complete and then returns the buffer header.

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

Managing the Buffer Cache briefly describes how the buffer cache services manage the block I/O buffer cache mechanism.

Execution Environment

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

Return Value

The **bread** service returns the address of the selected buffer's header.

Implementation Specifics

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

Related Information

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

Block I/O Buffer Cache Services: Overview, I/O Kernel Services in *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 block's 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.

Managing the Buffer Cache 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 Value

The **breada** service returns the address of the selected buffer's header.

Implementation Specifics

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

Related Information

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

The **ddstrategy** routine.

Block I/O Buffer Cache Services: Overview, I/O Kernel Services in *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

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

Related Information

The **geteblk** kernel service.

The buf structure.

Block I/O Buffer Cache Kernel Services: Overview, I/O Kernel Services, The buf Structure in *Kernel Extensions and Device Support Programming Concepts*.

bwrite Kernel Service

Purpose

Writes the specified buffer's 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's data. If this is a synchronous request, the **bwrite** service waits for the I/O to complete.

The article entitled Using the Buffer Cache write Services briefly describes how the three buffer cache write routines work.

Return Values

0	Indicates a successful operation.
Errno global variable	Indicates that an I/O error has occurred.

Execution Environment

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

Implementation Specifics

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

Related Information

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

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

cancel-queue-element Queue Management Routine

Purpose

Provides a means for performing cleanup of 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

Each device queue can have a **cancel-queue-element** routine. This routine is optional and must be specified when the device queue is created with the **creatq** service.

The **cancel-queue-element** routine is called by the kernel to clean up resources associated with a queue element. It 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 should be aborted.

The **cancel-queue-element** 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.

Related Information

The **creatq** kernel service, **canclq** kernel service, **peekq** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

canclq Kernel Service

Purpose

Deletes pending queue elements from a device queue.

Syntax

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

int canclq (*path_id*)
cba_id *path_id*;

Parameter

path_id	Specifies the path identifier.

Description

The **canclq** 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 once before the loading of any kernel extensions referencing these services.

The **canclq** service is intended for abnormal termination conditions. It allows a process to discard all pending queue elements on the specified path. The active queue element cannot be canceled.

Control-type queue elements are posted, and the server's **cancel-queue-element** queue management routine is called for each queue element canceled. This allows the server to abort any preprocessing of the request that the server initiated on a previous peek (using the **peekq** service) into the queue. It also allows the server to unpin memory associated with the request or to detach any cross-memory descriptors as appropriate. For a discussion of the device queue server and client model, see Understanding Device Queues.

Execution Environment

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

Return Value

The **canclq** service returns the number of canceled queue elements.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **peekq** kernel service.

The **cancel-queue-element** queue management routine.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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** config notification control block.

Description

The **cfgnadd** kernel service adds a **cfgncb** control block to the list of **cfgncb** structures maintained by the kernel. 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 effect the 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 **<sys/sysconfig.h>** file.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **sysconfig** subroutine.

The **cfgndel** kernel service.

The **cfgncb** configuration notification control block.

Kernel Program/Device Driver Management Kernel Services in *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.

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

The **cfgndel** or **cfgnadd** kernel services 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 **<sys/sysconfig.h>** file.

Notification Routine Calling Syntax

The **cfgncb.func** notification routine should be declared as follows:

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

Notification Routine Parameters

cmd	Indicates the current operation type. Possible values are CFGV_PREPARE and CFGV_COMMIT, as defined in the **<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.

Notification Routine Processing

Every time a SYS_SETPARMS **sysconfig** command is issued, the **sysconfig** subroutine iterates through the kernel's 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. If any of these values have changed, they 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 return with 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, sending the same parameters, except that the *cmd* parameter contains a value of CFGV_COMMIT. This indicates that the new values are to 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 verifies that the proposed new size for the cache is legal. On a CFGV_COMMIT command, the notification routine then makes the additional buffers available to the user (by chaining more buffers onto the existing list of buffers).

Related Information

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

The SYS_SETPARMS **sysconfig** Operation.

Kernel Program/Device Driver Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

cfgndel Kernel Service

Purpose

Removes a notification routine for receiving broadcasts of changes to system-configurable 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** 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 **sys/sysconfig.h** file contains a definition of the **cfgncb** structure.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **sysconfig** subroutine.

The **cfgnadd** kernel service.

The **cfgncb** configuration notification control block.

Kernel Program/Device Driver Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

check-parameters 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-parameters** 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-parameters** 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-parameters** 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 abort the request.

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

The address of the actual queue element is passed to this routine. In the **check-parameters** routine, take care to alter only the fields that were meant to be altered.

This routine typically does not need to be serialized with the rest of the server's routines, since it is just checking the parameters in the queue element.

The **check-parameters** routine can check the request before the request's 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-parameters** routine executes under the process environment of the requestor. 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.

Related Information

The **creatq** kernel service, **enque** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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 zeros the memory for the specified buffer structure's buffer.

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

Execution Environment

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

Implementation Specifics

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

Related Information

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

clrjmpx Kernel Service

Purpose

Removes a saved context by popping the most recently 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 be resumed, 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 the **setjmpx** service.

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

Execution Environment

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

Implementation Specifics

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

Related Information

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

Exception Processing, Implementing Exception Handlers, Process and Exception Management Kernel Services in *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 (*kaddr*, *uaddr*, *count*)
char **uaddr*;
char **kaddr*;
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 **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 halves 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.
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 invalid.

Implementation Specifics

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

Related Information

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

Accessing User-Mode Data While in Kernel Mode, Memory Kernel Services in *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	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 invalid.

Implementation Specifics

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

Related Information

Accessing User-Mode Data While in Kernel Mode, Memory Kernel Services in *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 ****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 **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 halves 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.
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 that the address specified in the *uaddr* parameter is invalid.

Implementation Specifics

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

Related Information

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

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

creatd Kernel Service

Purpose

Assigns a global name to a device queue.

Syntax

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

cba_id creatd (*iodn*, *queue_id*, *attach*, *detach*, *ptr*, *count*, *dev_parms*)
ushort *iodn*;
cba_id *queue_id*;
int (**attach*)();
int (**detach*)();
caddr_t *ptr*;
int *count*;
caddr_t *dev_parms*;

Parameters

iodn	Specifies the predetermined global name for the device queue. A value of DEFIND_PRIVATE indicates that no global name is required and the **queryd** service cannot be used to query the device identifier.
queue_id	Specifies the queue identifier of the device queue.
attach	Specifies the function pointer of the server's **attach-device** routine. The **attchq** service calls this routine when a new path to the device queue is created. This routine can have a NULL value if there is no device queue-specific processing to perform.
detach	Specifies the function pointer of the server's **detach-device** routine. The **detchq** service calls this routine when a path to the device queue is invalidated. This routine can have a NULL value if there is no device queue-specific processing to perform.
ptr	Specifies the address of the device-dependent information. The kernel enforces no format on this structure. The only purpose of this parameter is so that the **qryds** service can return a copy of this data to its caller. A NULL value indicates that there is no device-dependent information. If the *count* parameter is NULL, then this parameter must be 0 (zero)
count	Specifies the length of the device-dependent information. If the *ptr* parameter is NULL, then the *count* parameter must be 0.
dev_parms	Parameter passed to the device driver's device management routines.

Description

The **creatd** 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 once before loading any kernel extensions that reference these services.

The **creatd** service provides a means of associating a predefined global name (specified by the *iodn* parameter) with a device queue. The queue identifier and device identifier cannot be used for this purpose because their values cannot be predetermined. Additionally, device queue functions such as the automatic sending of a detach queue element are only performed if the requestor specified a device identifier when creating the path to the device queue.

The returned device identifier can be used to query information about the device using the **qryds** service. It can also be used to create a path to the associated device queue.

Note: The device being defined is associated with the process that is the server of the queue specified by the *queue_id* parameter.

The device queue host and client model is described in Understanding Device Queues.

Execution Environment

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

Return Values

Upon successful completion, the **creatd** service returns the new device identifier. This device identifier can be used when creating a path to the device queue. A value of NULL_CBA is returned in the following error cases:

- The value in the *iodn* parameter is already bound to a device queue.
- The queue identifier specified by the *queue_id* parameter is invalid.
- A control block could not be allocated.
- An error occurred during the cross-memory attach operation.
- The process was in the midst of termination.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **queryd** kernel service, **attchq** kernel service, **detchq** kernel service, **qryds** kernel service.

The **attach-device** queue management routine, **detach-device** queue management routine.

Understanding Device Queues, Device Queue Management Kernel Services in *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 the kernel process an identifier.
- Setting the process state to idle.
- Initializing its parent, child, and sibling relationships.

Kernel Process Creation, Execution, and Termination 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

Process Identifier	Indicates a successful operation.
–1	Indicates an error.

Implementation Specifics

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

Related Information

The **initp** kernel service.

Introduction to Kernel Processes, Process and Exception Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

creatq Kernel Service

Purpose

Creates a device queue.

Syntax

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

cba_id creatq (*server_id*, *worst_prty*, *max_path*, *max_qe*, *check*, *cancel*)
pid_t *server_id*;
uchar *worst_prty*;
uint *max_path*;
uint *max_qe*;
int (**check*)();
void (**cancel*)();

Parameters

server_id	Specifies the process identifier (PID) of the process that acts as server for this device queue.
worst_prty	Specifies the least favored queue element priority. The range of valid values is from QE_BEST_PRTY to QE_WORST_PRTY.
max_path	Specifies the maximum number of paths the device queue supports. The value must be in the range from 0 to MAX_QUEUE_PATH. A value of 1 indicates that only one process at a time can attach to the device queue. A value of 0 implies that there is no limit. Typically, this parameter is 1 for serially reusable devices and 0 otherwise, although other values can be specified.
max_qe	Specifies the maximum number of queue elements the device queue supports. This is the largest number of queue elements that can be waiting for service at any point in time. The value must be in the range from 0 to MAX_QE_DEPTH.
check	Specifies the function pointer to the server's **check-parameters** routine. This routine is called before a request queue element is placed on the device queue. A NULL value indicates that the server does not have a **check-parameters** routine.
cancel	Specifies the function pointer to the server's **cancel-queue-element** routine. This routine is called before a queue element is canceled. A NULL value indicates that the server does not have a cancel routine.

Description

The **creatq** 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 once before the loading of any kernel extensions referencing these services.

The **creatq** service can be used by any process to create a device queue. The new device queue can be served by the creating process or another process.

Each device queue served by a process has an event bit associated with it. This event bit is used to notify the process that the device queue is not empty. A unique event bit is assigned when a device queue served by a process is created. A queue's event bit cannot be used for any other purpose. The **e_post** service provides a description of event bit allocation. The event bit for a device queue can be determined by calling the **queryi** service.

There are two ways to determine if a device queue is not empty:

- The **e_wait** service can be called with one or more event bits, thus allowing a process to wait for input from one of multiple device queues.
- The **waitq** service can be called with a queue identifier.

Execution Environment

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

Return Values

Upon successful completion, the **creatq** service returns the device queue's identifier. The queue identifier is used as input to other services, such as the **deque** kernel service, to identify the device queue. If the device queue cannot be successfully created, a value of NULL_CBA is returned rather than the queue identifier.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **e_post** kernel service, **e_wait** kernel service, **queryi** kernel service, **waitq** kernel service.

The **check-parameters** queue management routine, **cancel-queue-element** queue management routine.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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 **<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 **<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** value and **time** value. The three in-memory time values that the kernel maintains (the **tod** value, **lbolt** value, and **time** value) 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 it 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.

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

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 the stack must be pinned when the **curtime** service is called in an interrupt handler environment.

Implementation Specifics

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

Related Information

Timer and Time-of-Day Kernel Services in *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. Cleaning up the DMA channel entails:

1. Marking the DMA channel specified by the *channel_id* parameter as free.
2. Resetting the DMA channel.

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

Warning: 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 system integrity compromised.

Execution Environment

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

Implementation Specifics

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

Related Information

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

Direct Memory Access (DMA), I/O Kernel Services in *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 **dma.h** header 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 successfully unhide 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.

DMA Transfer Modes and Block DMA Transfers further describe 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. An invalid or unsupported bus address was used. An invalid *daddr* parameter 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 data parity error. The system detected a data parity error.
DMA_ADDRESS	Indicates an address parity error. The system detected an 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

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

Related Information

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

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

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 **<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 (DDS) 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 consists of:

- 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

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

Related Information

The **d_clear** kernel service.

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

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	DMA channel identifier 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.

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

Note: The **d_mask** service, like all DMA services, should not be called unless the DMA channel has been allocated with the **d_init** service.

Execution Environment

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

Implementation Specifics

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

Related Information

The **d_init** kernel service, **d_unmask** kernel service.

Direct Memory Access (DMA), I/O Kernel Services in *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 **<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. Types of DMA Devices describes DMA slaves and masters.

The **d_master** service does not initiate the DMA transfer. The device initiates all DMA memory references. 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:

1. 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.
2. 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.
3. 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.

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 and remain inaccessible to the processor until the corresponding **d_complete** service has been issued once the pages are no longer required for DMA processing.

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 successfully unhide 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.

Note: 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.

Note: 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.

Note: 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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

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

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

d_move Kernel Service

Purpose

Provides consistent access to system memory that is accessed asynchronously by a device and by the processor on a RISC System/6000.

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 (zero) if the move is to be a write into system memory shared by a bus master device. The *flags* parameter should be set to DMA_READ if the move is to be a read from system memory shared by a bus master device. These flag values are defined in the **<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 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 master. The **d_move** service uses the same I/O controller data buffers that the DMA master does 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 RISC System/6000 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 (zero). A move of the data from the shared buffer to the nonshared buffer is effected 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.

Execution Environment

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

Return Values

XMEM_SUCC	Indicates successful completion.
XMEM_FAIL	Indicates one of these six 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

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

The **d_move** kernel service is available only on the RISC System/6000 product platform.

Related Information

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

Direct Memory Access (DMA), I/O Kernel Services in *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 **<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. Types of DMA Devices 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:

1. 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.
2. 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.
3. 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:

1. 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.
2. 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.
3. 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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **d_complete** kernel service, **d_init** kernel service, **xmattach** kernel service, **d_unmask** kernel service, **d_mask** kernel service.

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

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	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.

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

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.

Implementation Specifics

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

Related Information

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

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

del_arp_iftype Kernel Service

Purpose

Deletes an interface type from the Network ARP Switch Table Interface (NASTI).

Syntax

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

int del_arp_iftype(*if_type*, *af***)**
u_short *if_type*, *af*;

Parameters

if_type	Identifies the type of a network interface (for example, Ethernet or Token Ring). Interface types are defined in the **<sys/devinfo.h>** file.
af	Specifies the address family of the ARP routines being deleted.

Description

The **del_arp_iftype** kernel service deletes an interface type from the Network ARP Switch Table Interface (NASTI).

Example

1. The **del_arp_iftype** kernel service is invoked as follows:

   ```
   del_arp_iftype(DD_EN, AF_INET);
   ```

Return Values

0	Indicates that the interface was successfully deleted.
ENOENT	Indicates that the network type was not found for the specified address family.

Execution Environment

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

Implementation Specifics

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

Related Information

Network Kernel Services in *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

1. 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

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

Related Information

The **add_domain_af** kernel service.

Network Kernel Services in *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

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

Related Information

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

Network Kernel Services in *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 (zero) and less than a value of NETISR_MAX.

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

1. 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

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

Related Information

The **add_netisr** kernel service.

Network Kernel Services in *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.

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

Execution Environment

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

Implementation Specifics

This macro is part of AIX Base Operating System (BOS) Runtime.

Related Information

The **add_netopt** macro.

The **no** command.

Network Kernel Services in *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 **param.h** file can be used to determine the number of ticks per second.

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

Execution Environment

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

Implementation Specifics

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

Related Information

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

deque Kernel Service

Purpose

Performs completion processing for the active device queue element.

Syntax

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

int deque (*queue_id*, *options*, *qe*, *results***)**
cba_id *queue_id*;
int *options*;
struct ack_qe **qe*;
int *results*;

Parameters

queue_id	Specifies the identifier of the device queue from which to remove the active queue element.
options	Controls generation of the acknowledgment. The following values are possible:

SUPPRESS_ACK	Suppress acknowledgment.
OVERRIDE_VINTR	Override the virtual interrupt sublevel specified when the path was created.

qe	Specifies the address of the acknowledgment queue element or NULL.
results	Specifies the operation results for a synchronous request or an interrupt on error request.

Description

The **deque** 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 once before the loading of any kernel extensions referencing these services.

The **deque** service is called by a device queue server to tell the kernel that processing for the active queue element is complete. This service removes the active queue element from the device queue and conditionally sends an acknowledgment.

The **deque** service can automatically send an acknowledgment to the requestor if one was requested when the device queue was attached. Depending on the type of acknowledgment requested, different amounts of status information are returned. For simple interprocess communication, the acknowledgment functions are probably not necessary.

To generate an acknowledgment, the server of the device queue provides data in an acknowledgment queue element. The kernel then uses this data to send the acknowledgment to the requestor. The only time the acknowledgment queue element can be omitted is if the path type for the queue element is NO_ACK or SHORT_ACK or the suppress option SUPPRESS_ACK is selected.

The operation options field in the active queue element is examined to determine what operation the **deque** service should perform:

SYNC_REQUEST	Indicates that the operation is synchronous. On a SYNC_REQUEST, the **enque** routine enqueues the request and then sleeps, waiting for completion. When the **deque** service is called, it wakes up the **enque** routine and passes the results directly back. The **enque** service then passes the results to the caller.
ACK_COMPLETE	Indicates that an acknowledgment should be generated. If ACK_COMPLETE is specified, then an acknowledgment is sent each time a queue element is completed (dequeued) independent of the results.
ACK_ERRORS	Indicates that an acknowledgment should be generated only if there has been an error (the *results* parameter is not equal to RC_GOOD). If ACK_ERRORS is specified, the **deque** service only sends an acknowledgement on completed queue elements that have a result other than RC_GOOD.

These three operation flags are mutually exclusive. Therefore, only one should be specified.

If the suppress option is selected, the kernel does not return any information to the sender of the request. The device queue's server is responsible for explicitly generating the acknowledgment using the **ackque** service.

A path to a device queue may be destroyed before the active queue element is totally processed. If this happens, no acknowledgment is generated when the **deque** service is called. Instead, the queue element is discarded with no error reported.

Use of Virtual Interrupt Handlers

When an acknowledgment is sent through a path that was set up with an acknowledgment type of interrupt (INTR_ACK), then the **deque** service calls a registered virtual interrupt handler. This service uses the **qe–>data[5]** field in the acknowledgment queue element to provide a sublevel specifying which virtual interrupt handler to call. The sublevel specified when the path was created is used unless the OVERRIDE_VINTR value is specified in the *options* parameter to the **deque** service. Otherwise, the value in the acknowledgment queue element is used.

Virtual interrupt handlers can be registered by using the **vec_init** device queue management service. This interrupt handler is called in the process environment of the caller of the **deque** service. Virtual interrupts should be used for compatibility purposes only.

Execution Environment

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

Return Values

RC_GOOD	Indicates successful completion.
RC_OBJ	Indicates that there is no active queue element on the specified device queue.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **ackque** kernel service, **vec_init** kernel service, **enque** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *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 **detach** routine is called by the **detchq** service 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-device** routine can be called from the process environment only.

Return Values

RC_GOOD	Indicates successful completion.

A return value other than RC_GOOD indicates a fatal condition and causes the system to panic.

Related Information

The **creatd** kernel service, **detchq** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

detchq Kernel Service

Purpose

Invalidates the path to a device queue.

Syntax

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

int detchq (*path_id*)
cba_id *path_id*;

Parameter

path_id	Path identifier of the path to be invalidated.

Description

The **detchq** device queue 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 once before the loading of any kernel extensions referencing these services.

The **detchq** service invalidates the specified path.

If the **to_id** field in the path being invalidated is a device identifier, a detach queue element is placed in the device queue. The **detchq** service does not continue until the device queue server calls the **deque** service for this queue element. At this time, no other queue elements can be sent using this path. This serializes completion of all pending request for that path before invalidating the path.

However, this wait can also cause excessive delay for the caller of the **detchq** service if lengthy requests have yet to be processed. Device queue interfaces should be designed so that all I/O activity is finished before the **detchq** service is called. In addition, device queue servers must recognize detach queue elements. These detach queue elements are control queue elements sent by the kernel to detach a server from a path.

The server's **detach-device** queue routine is called if one is associated with the device queue. This occurs after the server calls the **deque** service for the detach queue element and executes under the caller of **detchq** process.

For device queues with multiple paths, a detach queue element is sent each time a path is invalidated.

Execution Environment

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

Return Values

RC_GOOD	Indicates successful completion.
RC_ID	Indicates that the path identifier is not valid.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **detach-device** queue management routine.

The **deque** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

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*, *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	A parameter or address to a parameter block for the specified command.
ext	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 error code 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 code for the **devdump** service is set to the return code 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.

dddump return codes
Return codes provided by the **dddump** device driver routine.

Implementation Specifics

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

Related Information

Device Switch Table.

The **dddump** Device Driver Entry Point.

Device Switch Table, Kernel Program/Device Driver Management Kernel Services in *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.

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.

Return Values

0	Indicates a successful operation.
ENODEV	Indicates 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

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

Related Information

The **iodone** kernel service.

The **ddstategy** routine.

Device Switch Table, The buf Structure, Kernel Program/Device Driver Management Kernel Services in *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 **devsw** (device switch) 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 the device switch structure is copied into the device switch table, 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 replaced, the **devswadd** service does not perform the update. Instead, it returns an EEXIST error code. If the update is successful, a value of 0 (zero) is returned.

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 that 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 routine, which is also provided by the kernel, performs no operation if called and returns a 0 return code.

All other fields within the structure that are not used should be set to 0 (zero). Some fields in the structure are for kernel use and are not copied into the device switch table by the **devswadd** service. These fields are documented in the **<sys/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

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

Related Information

The **devswdel** kernel service, **devswqry** kernel service.

The **ddconfig** Device Driver Entry Point.

Device Switch Table, Kernel Program/Device Driver Management Kernel Services in *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. 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 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 (zero) 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 **nodev**. 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

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

Related Information

The **devswadd** kernel service, **devswqry** kernel service.

Device Switch Table, Kernel Program/Device Driver Management Kernel Services in *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 address from the **d_dsdptr** field is not returned to the caller.

The status Parameter Flags

The *status* parameter comprises a set of flags that can indicate the following conditions:

DSW_UNDEFINED	Device switch entry is not defined.
DSW_DEFINED	Device switch entry is defined.
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_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_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_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 NULL.
DSW_SELECT	Device driver in this device switch entry is providing a routine for handling the **select** or **poll** subroutines. This flag is set when the device driver has provided a **ddselect** entry point.
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_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_OPENED	Device switch entry is in use and device has outstanding opens. This flag is set when the device driver has at least one outstanding open.

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.

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

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

Related Information

The **devswadd** kernel service, **devswdel** kernel service.

Device Switch Table, Kernel Program/Device Driver Management Kernel Services in *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 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 table 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 **<sys/dump.h>** header 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 **/usr/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.

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 **<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

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

Related Information

The **dmp_del** kernel service.

The **crash** command, **exec** command.

RAS Kernel Services in *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	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. If a kernel extension used the **dmp_add** service to register such a data area, it can use the **dmp_del** service to remove its 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

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

Related Information

The **dmp_add** kernel service.

The **crash** command.

RAS Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

dstryd Kernel Service

Purpose

Deletes a global name from a device queue.

Syntax

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

int dstryd (*device_id***)**
cba_id *device_id*;

Parameter

device_id	Device identifier of the device queue.

Description

The **dstryd** device queue 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 once before loading any kernel extensions that reference these services.

The **dstryd** service provides a means of deleting a device ID, specified by the *device_id* parameter, from a device queue. This device identifier was previously returned by the **creatd** device queue management service when the device queue was assigned a global name.

The device is removed from the global system list of valid devices. The device identifier is also removed from the process's list of devices. The cross-memory descriptor is then detached and the device control block is freed.

Execution Environment

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

Return Values

RC_GOOD	Indicates that the the ID specified by the *device_id* parameter is not valid (that is, no longer exists) or that the operation was successful.
RC_IN_USE	Indicates that the device is still being used.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **creatd** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

dstryq Kernel Service

Purpose

Destroys the specified device queue.

Syntax

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

int dstryq (*queue_id***)**
cba_id *queue_id*;

Parameter

queue_id	Identifies the device queue to destroy.

Description

The **dstryq** 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 once before the loading of any kernel extensions referencing these services.

The **dstryq** service is used to destroy a device queue. Any process can destroy any device queue.

The queue elements in a device queue, if any, are discarded when the device queue is destroyed. Acknowledgments are not generated when the queue elements are discarded. The server of the device queue is not informed that the device queue was destroyed, unless it was waiting on the device queue. If the process was waiting, it is returned to the ready state with the event posted. Also, any paths to the device queue are destroyed.

The server's **cancel-queue-element** routine is called for each queue element, active or pending, that is discarded from the device queue.

Execution Environment

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

Return Values

RC_GOOD	Indicates successful completion.
RC_ID	Indicates that the device queue identifier is not valid.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **cancel-queue-element** queue management routine.

Understanding Device Queues, Device Queue Management Kernel Services in *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/types.h>
#include <sys/errno.h>
#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 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, in that the input parameter can point to any address within the **mbuf** structure, not merely the address of the actual data.

Example

1. The **DTOM** macro can be used as in the following example:

```
char            *bp;
struct mbuf     *m;
m = DTOM(bp);
```

Implementation Specifics

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

Related Information

MTOD macro for **mbuf** Kernel Services.

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

e_post Kernel Service

Purpose

Notifies a process of the occurrence of one or more events.

Syntax

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

void e_post (*events*, *pid*)
unsigned long *events*;
pid_t *pid*;

Parameters

events	Identifies the masks of events to be posted.
pid	Specifies the process identifier of the process to be notified.

Description

The **e_post** kernel service is used to notify a process that one or more events occurred. The **e_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 **e_post** service is performed automatically when sending a request to a device queue serviced by a process 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 (one) indicates an event bit reserved for the kernel. Kernel extensions should assign their events starting with the most significant bits and working down. If processes using the **e_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 **e_wait** service does not sleep but returns immediately if a specified event has already been posted by the **e_post** service.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **e_wait** kernel service.

Understanding Device Queues, Process and Exception Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

e_sleep Kernel Service

Purpose

Forces a process 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 **<sys/sleep.h>** file.

Description

The **e_sleep** kernel service is used to wait for the specified shared event to occur. The kernel places the current process on the list anchored by the *event_word* parameter. This list is used by the **e_wakeup** service to wake up all processes 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 process 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 **e_sleep** for the event, the process still sleeps, waiting on the next **e_wakeup** 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** for the event has been called.

Flags

The *flags* parameter is used to control how signals affect waiting for an event.

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_SIGRET	Flag overrides the **EVENT_SIGWAKE** flag.
EVENT_SHORT	Prohibits the wait from being terminated by a signal. This flag should only be used for short, guaranteed-to-wakeup sleeps.

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

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

Related Information

The **e_sleepl** kernel service, **e_wakeup** kernel service.

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

e_sleepl Kernel Service

Purpose

Forces a process 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 processes 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 **<sys/sleep.h>** file.

Description

The **e_sleepl** kernel service waits for the specified shared event to occur. The kernel places the current process on the list anchored by the *event_word* parameter. The **e_wakeup** service wakes up all processes on the list.

The **e_wakeup** service does not wake up a process that is not currently sleeping in the **e_sleepl** function. That is, if an **e_wakeup** operation for an event is issued before the process calls **e_sleepl** for the event, the process still sleeps, waiting on the next **e_wakeup** 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** for the event has been called.

The **e_sleepl** service also unlocks the conventional lock specified by the *lock_word* parameter before putting the process to sleep. It also reacquires the lock when the process 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.

Flags

The *flags* parameter controls how signals affect waiting for an event. There are three flags available to the **e_sleepl** service: **EVENT_SIGRET, EVENT_SIGWAKE,** and **EVENT_SHORT.**

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.

Note: The **EVENT_SIGRET** flag overrides the **EVENT_SIGWAKE** flag.

EVENT_SHORT
Indicates that signals cannot terminate the wait. Use the **EVENT_SHORT** flag for only short, guaranteed-to-wakeup sleeps.

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

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

Related Information

The **e_sleep** kernel service, **e_wakeup** kernel service.

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

e_wait Kernel Service

Purpose

Forces a process to wait for the occurrence of an event.

Syntax

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

ulong e_wait (*wait_mask*, *clear_mask*, *flags***)**
ulong *wait_mask*;
ulong *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 that control actions on occurrence of a signal.

Description

With the **e_wait** kernel service, a process waits for any of one or more specified events. The *wait_mask* parameter is a mask, where each bit set equal to 1 (one) represents an event to wait for.

The *clear_mask* parameter is a mask of events to clear when the wait is complete. Subsequent calls to the **e_wait** service return immediately unless you clear the bits, which terminates the wait.

The **e_wait** service can also be used to clear events without waiting for them to occur. This is accomplished by setting:

- The *wait_mask* parameter to EVENT_NDELAY
- The bits in the *clear_mask* parameter that correspond to the events to be cleared to a value of 1 (one).

This form can also be used to poll the events because an event mask indicating those events that were actually cleared is returned by the **e_wait** service.

Flags

The *flags* parameter is used to control how signals affect waiting for an event. There are two flag values: **EVENT_SIGRET** and **EVENT_SIGWAKE**.

EVENT_SIGRET
Causes the wait for the event to be terminated by an unmasked signal and the return value to be set to EVENT_SIG.

EVENT_SIGWAKE
Causes the event to be terminated by an unmasked signal and control transferred to the return from the last **setjmpx** call, with the return value set to EXSIG.

Note: The **EVENT_SIGRET** flag overrides the **EVENT_SIGWAKE** flag.

Execution Environment

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

Return Values

Upon successful completion, the **e_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.

EVENT_SIG	Indicates that the **EVENT_SIGRET** flag is set and the wait is terminated by a signal.

Implementation Specifics

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

Related Information

The **e_post** kernel service, **setjmpx** kernel service.

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

e_wakeup Kernel Service

Purpose

Notifies processes waiting on a shared event of the event's occurrence.

Syntax

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

void e_wakeup (*event_word***)**
int **event_word*;

Parameter

event_word	Specifies the shared event designator. The kernel uses the *event_word* parameter as the anchor to the list of processes waiting on this shared event.

Description

The **e_wakeup** kernel service is used to notify the list of processes anchored by the *event_word* parameter that the event has occurred. The anchor for the event list, specified by the *event_word* parameter, is set to EVENT_NULL by the **e_wakeup** service.

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

Execution Environment

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

When called by an interrupt handler, the *event_word* parameter must be located in pinned memory.

Implementation Specifics

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

Related Information

The **e_sleep** kernel service, **e_sleepl** kernel service.

Process and Exception Management Kernel Services in *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 once before the loading of 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 requestor 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:

SEND_CMD	Send Command.
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 **e_wait** or **waitq** kernel services.

The kernel calls the server's **check-parameters** routine, if one is defined, before a queue element is placed on the device queue. This routine can abort 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 **e_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

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **e_wait** kernel service, **waitq** kernel service, **e_post** kernel service.

The **check-parameters** queue management routine.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

errsave 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*;

Parameters

buf	Points to a buffer that contains an error record as described in the **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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

RAS Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

find_arp_iftype Kernel Service

Purpose

Finds an interface type in the Network ARP Switch Table Interface (NASTI).

Syntax

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

struct nasti_ent *find_arp_iftype (*if_type*, *af*)
u_short *if_type*, *af*;

Parameters

if_type	Identifies the type of a network interface (for example, Ethernet or token ring). Interface types are defined in the **/usr/include/sys/devinfo.h** file.
af	Specifies the address family of the ARP routines being deleted.

Description

The **find_arp_iftype** kernel service finds an interface type in the Network ARP Switch Table Interface (NASTI). If successful, a **nasti_ent** structure is returned that contains the following fields:

if_type	Uniquely identifies the type of a network interface (for example, Ethernet or token ring). Interface types are defined in the **/usr/include/sys/devinfo.h** file.
af	Specifies the address family that the specified ARP routines are able to handle.
ioctl	Points to an ARP **ioctl** handler.
resolve	Points to an ARP resolve handler.
whohas	Points to a function for transmitting ARP request packets.
arptfree	Points to a function that frees ARP entries and reclaims resources.

Example

1. The **find_arp_iftype** kernel service is invoked as follows:

```
find_arp_iftype(DD_EN, AF_INET);
```

Return Values

Pointer to a nasti_ent structure	Indicates that the requested interface type was found.
NULL	Indicates that the type of network indicated by the *if_type* parameter cannot be found.

Execution Environment

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

Implementation Specifics

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

Related Information

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

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*, *samp_flags*, *mh***);**
ushort *type*;
struct mbuf **m*;
struct arpcom **ac*;
ulong *snmp_flags*;
struct mbuf **mh*;

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 in the **netinet/in_netarp.h** file.
snmp_flags	Specifies either the **UCAST_TYPE** or **NUCAST_TYPE** flags. This parameter is set to the **NUCAST_TYPE** flag if the packet was a broadcast. Otherwise, the **NUCAST_TYPE** flag is set.
mh	Points to an **mbuf** buffer containing the input packet header. Protocols that ask for packet headers will receive a copy of *mh* prepended to *m*.

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* parameter. 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

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

Related Information

The **add_input_type** kernel service, **del_input_type** kernel service.

Network Kernel Services in *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>

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 services.
perm	Indicates which read, write, and execute permissions are to be checked. The **<sys/access.h>** file contains pertinent values.

Description

The **fp_access** kernel service is an internal interface to the function provided by the **access** subroutine.

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

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

Related Information

The **access** subroutine.

Logical File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

fp_close Kernel Service

Purpose

Closes a file.

Syntax

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

fp_close (*fp***)**
struct file **fp*;

Parameter

fp	Points to a file structure returned by the **fp_open**, **fp_getf**, **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 **<sys/error.h>** file is returned.

Implementation Specifics

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

Related Information

The **close** subroutine.

Logical File System Kernel Services in *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>

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** type structure or **fullstat** structure. The *statsz* parameter indicates the buffer type.
statsz	Indicates the size of the **stat** structure to be returned (that is, **stat** or **fullstat**). The **<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** system call.

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 **<sys/errno.h>** file is returned.

Implementation Specifics

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

Related Information

The **fstatx** subroutine.

Logical File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

fp_getdevno Kernel Service

Purpose

Gets the device number and/or channel number for a device.

Syntax

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

fp_getdevno (*fp*, *devp*, *chanp***)**
**struct file ****fp*;
**dev_t ****devp*;
**chan_t ****chanp*;

Parameters

fp	Points to a file struct returned by the **fp_open** or **fp_opendev** services.
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 *devp* or *chanp* parameter is specified as NULL, this service will 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 struct for an open device.

Implementation Specifics

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

Related Information

Logical File System Kernel Services in *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>

fp_getf (*fd*, *fpp*)
int *fd*;
struct file ***fpp*;

Parameters

fd	Specifies a file descriptor.
fpp	Points to the location where the file structure pointer is to be returned.

Description

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.

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.

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 out of the range of valid file descriptors or the file descriptor is not currently used in the process.

Implementation Specifics

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

Related Information

Logical File System Kernel Services in *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

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

Related Information

Logical File System Kernel Services in *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>

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 **<sys/errno.h>** file is returned. The **ioctl** subroutine contains valid **errno** values.

Implementation Specifics

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

Related Information

The **ioctl** subroutine.

Logical File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

fp_lseek Kernel Service

Purpose

Changes the current offset in an open file.

Syntax

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

fp_lseek (*fp*, *offset*, *whence*)
struct file **fp*;
long *offset*;
unsigned 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** kernel service is an internal interface to the function provided by the **lseek** subroutine.

Execution Environment

The **fp_lseek** 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 **<sys/errno.h>** file is returned.

Implementation Specifics

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

Related Information

The **lseek** subroutine.

Logical File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

fp_open Kernel Service

Purpose

Opens a regular file or directory.

Syntax

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

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. Its content, form, and use are determined by the individual driver.
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 following:

- 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 **<sys/errno.h>** file is returned. The discussion of the **open** subroutine contains possible **errno** values.

Implementation Specifics

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

Related Information

The **open** subroutine.

Logical File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

fp_opendev Kernel Service

Purpose

Opens a device special file.

Syntax

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

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 character string or NULL.
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 **fp_opendev** kernel service is called by the kernel or kernel extension 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 inode is found or created for this device.
- The inode 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 (zero).

If the device driver is a multiplexed character device driver (that is, its **ddmpx** entry point is defined), an in-core inode 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 inode 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 code 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 zero, the file pointer corresponding to this open of the device is returned.

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.

A file pointer (specified by the *fpp* parameter) 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 **<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.
ddopen return codes	Any nonzero return code returned from a device driver **ddopen** routine.

Implementation Specifics

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

Related Information

Logical File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

fp_poll Kernel Service

Purpose

Checks the I/O status of multiple file pointers/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/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 (zero), then the corresponding array is presumed absent.
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 (zero), 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 **<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 (zero), then the *listptr* parameter must point to an array of **pollmsg** structures.

If the number of **pollmsg** elements in the *nfdsmsgs* parameter is 0 (zero), 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.

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

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

Related Information

The **poll** subroutine.

Logical File System Kernel Services in *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>

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 **<sys/errno.h>** file is returned.

Implementation Specifics

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

Related Information

The **read** subroutine.

Logical File System Kernel Services in *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>

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 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 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 **<sys/errno.h>** file is returned.

Implementation Specifics

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

Related Information

The **readv** subroutine.

Logical File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

fp_rwuio Kernel Service

Purpose

Performs read and write operations on an open file with arguments passed in a **uio** structure.

Syntax

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

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 the 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 **<sys/errno.h>** file is returned.

Implementation Specifics

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

Related Information

The uio Structure, Logical File System Kernel Services in *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 **<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

The **fp_select** kernel service can be called by a device driver's **ddselect** routine 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 may 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.
Nonzero return code from the ddselect routine	Return code provided by the specified object's **ddselect** routine. The *rtneventp* parameter has the **POLLERR** flag set.

Implementation Specifics

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

Related Information

The **selnotify** kernel service, **fp_poll** kernel service.

The **select** subroutine, **poll** subroutine.

The **fp_select notify** routine.

Logical File System Kernel Services in *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>

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.

If an error occurs, one of the values from the **<sys/errno.h>** file is returned.

Implementation Specifics

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

Related Information

The **write** subroutine.

Logical File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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>

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 in which part of memory the information designated by the *iov* parameter is located:

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 **<sys/errno.h>** file is returned.

Implementation Specifics

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

Related Information

The **writev** subroutine.

Logical File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

fubyte Kernel Service

Purpose

Fetches, or 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 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

The specified byte	Indicates successful completion.
–1	Indicates a *uaddr* parameter that is invalid.

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

This kernel service is part of AIX 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, Memory Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

fuword Kernel Service

Purpose

Fetches, or 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

The specified word	Indicates successful completion.
–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.

For the **fuword** service, a retrieved value of –1 and a –1 return code are indistinguishable.

Implementation Specifics

This kernel service is part of AIX 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, Memory Kernel Services in *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/addspace.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 Value

The **getadsp** service returns a pointer to the current process's address space structure.

Implementation Specifics

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

Related Information

The **as_att** kernel service, **as_det** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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, then 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, then another buffer is taken from the free list and returned.

Managing the Buffer Cache summarizes how the **getblk, bread**, and **brelse** services uniquely manage the block I/O buffer cache.

Execution Environment

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

Return Value

The **getblk** service returns a pointer to the buffer header. There are no error codes because the **getblk** service waits until a buffer header becomes available.

Implementation Specifics

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

Related Information

Block I/O Buffer Cache Services: Overview, I/O Kernel Services in *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

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.

Warning: 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.

Execution Environment

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

Return Value

–1	Indicates that the character list is empty.

Implementation Specifics

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

Related Information

I/O Kernel Services in *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

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.

Warning: 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.

Execution Environment

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

Return Values

NULL address	Indicates that 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

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

Related Information

I/O Kernel Services in *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

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.

Warning: 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.

Execution Environment

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

Return Value

The **getcbp** service returns the number of characters retrieved from the character buffer.

Implementation Specifics

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

Related Information

I/O Kernel Services in *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 invoking the **getcf** service, the caller should request enough **clist** resources by using the **pincf** kernel service. The proper use of the **getcf** service insures 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.

NULL pointer	Indicates that no buffers are available.

Implementation Specifics

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

Related Information

I/O Kernel Services in *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

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.

Note: 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.

Execution Environment

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

Return Value

The **getcx** service returns the character at the end of the list instead of the character at the front of the list.

Implementation Specifics

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

Related Information

I/O Kernel Services in *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

Note: 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. For a comparison of block and character device drivers, see Comparison of Block and Character Device Drivers.

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. (For example, 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

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

Related Information

The **brelse** kernel service, **xmalloc** kernel service.

Introduction to Kernel Buffers, Block I/O Buffer Cache Services: Overview, Physical Device Support, I/O Kernel Services in *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 it is not set, the **geterror** service returns 0 (zero). Otherwise, 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

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

Related Information

Block I/O Buffer Cache Services: Overview, I/O Kernel Services in *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 **sys/except.h** header 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 **sys/except.h** header 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.

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

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.

Implementation Specifics

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

Related Information

Non-Recoverable Hardware I/O Errors, Kernel Program/Device Driver Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

getpid Kernel Service

Purpose

Gets the process ID of the current process.

Syntax

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

int 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

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

Related Information

Understanding Execution Environments, Process and Exception Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

getuerror Kernel Service

Purpose

Allows kernel extensions to retrieve the current value of the **u_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 process's **u_error** field. Kernel extensions can use the **getuerror** service when using system calls or other kernel services that return error information in the **u_error** field.

For system calls, the system call handler copies the value of the **u_error** field in the per process **u** block 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 **u_error** in the per process **u** block. Possible return values for this field are defined in the **<sys/errno.h>** header file.

Implementation Specifics

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

Related Information

The **setuerror** kernel service.

Actions of the System Call Handler describes the steps that the system call handler takes when a system call is invoked in user mode.

Returning Error Information describes how system calls return error information.

Understanding System Call Execution, Returning Error Information from System Calls, Kernel Program/Device Driver Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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 is a small integer value, either one of those defined in the **<sys/vmount.h>** header 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 **<sys/gfs.h>** header 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 **<sys/vmount.h>** header 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 indicating whether the file system uses System V-type directories and whether it is a distributed file system.
gfs_ops	Specifies the array of pointers to **vfs** operation implementations.
gn_ops	Specifies the array of pointers to vnode 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 **<sys/vmount.h>** header file.

Implementation Specifics

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

Related Information

The **gfsdel** kernel service.

Understanding Data Structures and Header Files for Virtual File Systems, Virtual File System Overview, Virtual File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

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 invalid 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 **<sys/vmount.h>** header file.
EBUSY	Indicates that there are active **vfs** structures for the file system type being deleted.

Implementation Specifics

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

Related Information

The **gfsadd** kernel service.

Virtual File System Overview, Virtual File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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 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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **i_init** kernel service.

Processing Interrupts.

Understanding Interrupts, Processing Interrupts, I/O Kernel Services in *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

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.

Note: The **i_disable** service is very similar to the standard UNIX **spl** service.

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 **<sys/intr.h>** header file defines the 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. Coding an Interrupt Handler contains a brief description of interrupt handlers.

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.

Warning: The **i_disable** service has two side effects that result from the preemptable and pageable nature of the AIX 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.

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 1Kbyte.
- The called programs use less than 1Kbyte of stack.

The caller of the **i_disable** service should also call only services that can be called by interrupt handlers.

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. The interrupt priority 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 used subsequently with the **i_enable** service.

Implementation Specifics

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

Related Information

The **i_enable** kernel service, **i_mask** kernel service.

Processing Interrupts.

Understanding Interrupts, Processing Interrupts, I/O Kernel Services in *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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **i_disable** kernel service.

Processing Interrupts.

Understanding Interrupts, Processing Interrupts, I/O Kernel Services in *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

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 **<sys/intr.h>** header 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.

Warning: 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. It must also stay pinned. If this structure is altered at those times, a kernel panic may result.

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 then 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 Note that 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. More details on this are provided in Early Power–Off Warning discussion in Processing Interrupts

The return code 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 either the process or interrupt environment.

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

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

Related Information

Processing Interrupts.

Early Power–Off Warning discussion in Processing Interrupts.

Understanding Interrupts, Processing Interrupts, I/O Kernel Services in *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 them when they must allow process execution with their device interrupts disabled.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **i_unmask** kernel service.

Processing Interrupts.

Understanding Interrupts, Processing Interrupts, I/O Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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. Coding an Interrupt Handler contains a brief description of interrupt handlers.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **i_init** kernel service.

Processing Interrupts.

Understanding Interrupts, Processing Interrupts, I/O Kernel Services in *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. Coding an Interrupt Handler contains a brief description of interrupt handlers.

Processing can be scheduled off-level in the following situation:

- The interrupt handler routine for a device driver must perform time-consuming processing.
- This work does not need to be performed immediately.

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 **<sys/intr.h>** header 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.

Notes:

1. 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. It must also stay pinned. Otherwise, the system may crash.
2. Off-level interrupt handler path length should not exceed 5,000 instructions. If it does exceed this number, real-time support is adversely affected.The **i_sched** service has no return values.

Return Values

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **i_init** kernel service.

Processing Interrupts.

Understanding Interrupts, Processing Interrupts, I/O Kernel Services in *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. The **i_unmask** service has no return values.

Execution Environment

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

Implementation Specifics

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

Related Information

The **i_init** kernel service, **i_mask** kernel service.

Processing Interrupts.

Understanding Interrupts, Processing Interrupts, I/O Kernel Services in *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. The **if_attach** kernel service has no return values.

Execution Environment

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

Implementation Specifics

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

Related Information

The **if_detach** kernel service.

Network Kernel Services in *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

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

Related Information

The **if_attach** kernel service.

Network Kernel Services in *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 **ifnet** structure **flags** field as not up
- Notifies the protocols of the transaction
- Flushes the output queue.

The *ifp* parameter specifies the **ifnet** structure associated with the interface to be marked as down.

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

Execution Environment

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

Example

To mark an interface as down, invoke the **if_down** kernel service as follows:

```
if_down(ifp);
```

Implementation Specifics

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

Related Information

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

if_nostat Kernel Service

Purpose

Zeros 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 zeros 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.

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

Execution Environment

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

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

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

Related Information

Network Kernel Services in *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_withaddr** 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.

Example

1. To locate an interface based on a complete address, invoke the **ifa_ifwithaddr** kernel service as follows:

```
ifa_ifwithaddr((struct sockaddr *)&ipaddr);
```

Return Values

If successful, the **ifa_withaddr** service returns the corresponding **ifaddr** structure associated with the address it is passed. If no interface is found, the **ifa_withaddr** service returns a NULL pointer.

Implementation Specifics

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

Related Information

The **ifa_ifwithdstaddr** kernel service, **ifa_ifwithnet** kernel service.

Network Kernel Services in *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.

Example

1. 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);
```

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.

Implementation Specifics

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

Related Information

The **ifa_ifwithaddr** kernel service, **ifa_ifwithnet** kernel service.

Network Kernel Services in *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. If successful, the **ifa_ifwithnet** service returns the **ifaddr** structure of the correct interface.

Execution Environment

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

Example

1. To locate an interface on a specific network, invoke the **ifa_ifwithnet** kernel service as follows:

```
ifa_ifwithnet((struct sockaddr *)&ipaddr);
```

Return Values

If successful, the **ifa_ifwithnet** service returns the corresponding **ifaddr** structure associated with an interface. If no interface is found, the **ifa_ifwithnet** service returns a NULL pointer.

Implementation Specifics

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

Related Information

The **ifa_ifwithaddr** kernel service, **ifa_ifwithdstaddr** kernel service.

Network Kernel Services in *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.

Example

1. To return a pointer to the **ifnet** structure of the requested interface, invoke the **ifunit** kernel service as follows:

```
ifp = ifunit("en0");
```

Return Values

The **ifunit** kernel service returns the address of the **ifnet** structure associated with the named interface.

NULL	Indicates that the interface was not found.

Implementation Specifics

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

Related Information

Network Kernel Services in *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>

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 value of NULL. This field is used by the base kernel 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 a private heap is created with this service, 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 may 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

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

Related Information

The **xmalloc** kernel service, **xmfree** kernel service.

Introduction to Kernel Processes, Memory Kernel Services in *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***) ( )**
**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.

Kernel Process Creation, Execution, and Termination further explains how the **initp** kernel service completes the initialization process begun by the **creatp** service.

Description of Parameters

The *pid* parameter identifies the process to be initialized. It must be valid and identify a process in the **SIDL** 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.

The subroutine defined by the *func* parameter can be declared as follows:

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

void *func* (*flag*, *init_parms*, *parms_length*)
int *flag*;
void **init_parms*;
int *parms_length*;

Where the *flag* parameter has a 0 (zero) value if this subroutine is executed as a result of initializing a process with the **initp** service.

Execution Environment

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

Return Values

0	Indicates a successful operation.
ENOMEM	Indicates that there was insufficient memory to initialize the process.
EINVAL	Indicates an invalid *pid* parameter.

Implementation Specifics

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

Related Information

The **creatp** kernel service.

Introduction to Kernel Processes, Process and Exception Management Kernel Services in *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

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.

Warning: The **io_att** service will crash the kernel if there are no more free regions.

Return Value

address	Indicates the address for offset in the virtual memory address space.

Execution Environment

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

Implementation Specifics

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

Related Information

The **io_det** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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/addspace.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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **io_att** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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 invoked 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.

Return Values

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **iowait** kernel service.

The **buf** structure.

Block I/O Processing, Understanding Block I/O Device Drivers, The buf Structure, Understanding Interrupts, I/O Kernel Services in *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 **<sys/devinfo.h>** header file. Currently, I/O statistics are only kept for disks, CDROMs, and TTY devices. Possible values for this parameter currently are:

DD_DISK	for disks.
DD_CDROM	for CDROMs.
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_CDROM, 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 CDROM 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 device configured.

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 CDROM devices with a *devtype* value of DD_DISK or DD_CDROM, 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 de-registered by calling the **iostdel** kernel service.

I/O Statistics Structures

The **iostadd** kernel service uses two structures of interest that are found in the **<sys/iostat.h>** header file. The **ttystat** structure contains the following TTY-related fields:

rawinch	A count of raw characters received by the TTY device.
caninch	A count of canonical characters generated from canonical processing.
outch	A 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	A 32-character string name for the disk's logical device.
dknextp	A pointer to the next **dkstat** structure in the chain.
dk_status	Disk entry status flags.
dk_time	The time the disk is active.
dk_bsize	The number of bytes in a block.
dk_xfers	The number of transfers to or from the disk.
dk_rblks	The number of blocks read from the disk.
dk_wblks	The number of blocks written to the disk.
dk_seeks	The 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 (zero).

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 (Kbytes/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 **dkstat.dk_nextp** and **dkstat.dk_time** fields are updated by the base kernel and 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 an invalid device type was specified by the *devtype* parameter.

Implementation Specifics

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

Related Information

The **iostdel** kernel service.

The **iostat** command.

Kernel Extension/Device Driver Management Kernel Services in *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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **iostadd** kernel service.

The **iostat** command.

Kernel Extension/Device Driver Management Kernel Services in *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 with 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 these two values to return:

0	Indicates that I/O was successful on this buffer.
EIO or the **b_error** value in the **buf** header	Indicates that an I/O error has occurred.

Implementation Specifics

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

Related Information

The **iodone** kernel service, **geterror** kernel service.

The buf structure.

Block I/O Processing, Understanding Block I/O Device Drivers, The buf Structure, I/O Kernel Services in *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

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

Related Information

The **sethostname** subroutine.

Network Kernel Services in *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

non–NULL function pointer	Indicates a successful completion. This function pointer contains the module's entry point.
NULL function pointer	Indicates an error.

Implementation Specifics

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

Related Information

Kernel Program/Device Driver Management Kernel Services in *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 /**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 libpath 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 invalid if the **kmod_load** service returns a nonzero return code.

Description

The **kmod_load** kernel service loads a kernel extension object file specified by a path name into the kernel. This service returns a kernel module ID for that instance of the module.

Flags can be specified 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 libpath 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 libpath search string is used to find the location of the shared object module required to resolve the import. This libpath 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 libpath to be used. However, if the *libpathp* parameter is NULL, then the libpath is to be taken from the module header for the object module specified by the *pathp* parameter.

The libpath 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.

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 invalid XCOFF indicator 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

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

Related Information

The **kmod_unload** kernel service.

Kernel Program/Device Driver Management Kernel Services in *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 flags are defined:

0	Unload 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. The 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 invalid or does not correspond to a currently loaded module.

Implementation Specifics

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

Related Information

The **kmod_load** kernel service.

Kernel Program/Device Driver Management Kernel Services in *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 **<sys/msg.h>** header 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 members 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** member, 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 that: 1) the identifier specified by the *msqid* parameter is not a valid message queue identifier or that 2) 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** member 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** member, but the calling process does not have the appropriate system privilege.

Implementation Specifics

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

Related Information

The **msgctl** subroutine.

The User Protection Domain, Kernel Protection Domain, Kernel-Mode Message Kernel Services in *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 **<sys/stat.h>** header 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 comand 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

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

Related Information

The **msgget** subroutine.

The User Protection Domain, Kernel Protection Domain, Kernel-Mode Message Kernel Services in *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 **sys/msg.h** header 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. 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.
msgtyp	Specifies the type of message requested as follows: • If the *msgtyp* parameter is equal to 0 (zero), 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 a 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.

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 (zero).
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

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

Related Information

The **msgrcv** subroutine, **msgxrcv** subroutine.

The User Protection Domain, Kernel Protection Domain, Kernel-Mode Message Kernel Services in *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*, *msgz*, *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 **<sys/msg.h>** header file.
msgz	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 in the manner prescribed 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

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

Related Information

The **msgsnd** subroutine.

The User Protection Domain, Kernel Protection Domain, Kernel-Mode Message Kernel Services in *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.

Description

The **lockl** kernel service locks a conventional lock. The kernel provides conventional locks as a means of accessing and updating global memory in a preemptable kernel.

The kernel uses the **lockl** service to control access to shared memory. The lock word can also 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 variable must be initialized to the LOCK_AVAIL value before the first call to the **lockl** service. It must not be altered except by the **lockl** and **unlockl** services 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. Preempting a System Call discusses how system calls can use the **lockl** service when accessing global data.

Flags

The *flags* parameter is used to control how signals affect waiting for a lock. There are three flags: **LOCK_SIGRET**, **LOCK_SIGWAKE**, and **LOCK_NDELAY**.

LOCK_SIGRET Causes the wait for the lock to be terminated by an unmasked signal.

Note: The **LOCK_SIGRET** flag overrides the **LOCK_SIGWAKE** flag.

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.
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.

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 *flag* parameter is set to **LOCK_NDELAY**.
LOCK_FAIL	Indicates that the lock is owned by another process when the *flag* parameter is set to **LOCK_NDELAY**.
LOCK_SIG	Indicates that the wait is terminated by a signal when the *flag* parameter is set to **LOCK_SIGWAKE**.

Implementation Specifics

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

Related Information

The **unlockl** kernel service.

Pre-empting a System Call, Understanding Locking, Understanding Signal and Exception Handling, Process and Exception Management Kernel Services in *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 Value

The **loifp** service returns the address of the **ifnet** structure describing the software loopback interface.

Implementation Specifics

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

Related Information

The **looutput** kernel service.

Network Kernel Services in *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***;**

label_t ****jump_buffer*;**

Parameters

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

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

Related Information

The **setjmpx** kernel service, **clrjmpx** kernel service.

Exception Processing, Implementing Exception Handlers, Process and Exception Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

lookupvp Kernel Service

Purpose

Retrieves the vnode that corresponds to the named path.

Syntax

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

int lookupvp (*namep, flags, vpp***)**
char **namep;*
int *flags;*
struct vnode ***vpp;*

Parameters

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 vnode 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 pathname 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 vnode 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 vnodes 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 **<sys/errno.h>** header file.

Implementation Specifics

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

Related Information

Understanding Data Structures and Header Files for Virtual File Systems, Virtual File System Overview, Virtual File System Kernel Services in *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

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

Related Information

The **loifp** kernel service.

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

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 non-negative, the bytes are removed from the front of the chain. If this number is negative, the alteration is done from back to front.

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

Execution Environment

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

Implementation Specifics

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

Related Information

I/O Kernel Services in *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 specifed by the *n* parameter to the end of **mbuf** chain specified by the *m* parameter. Where possible, compaction is performed.

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

Execution Environment

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

Implementation Specifics

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

Related Information

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

m_clget Kernel Service

Purpose

Allocates a page-sized **mbuf** structure cluster.

Syntax

#include <sys/types.h>
#include <sys/errno.h>
#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** kernel service 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** kernel service can be called from either the process or interrupt environment.

Return Values

1	Indicates successful completion.
0	Indicates an error.

Implementation Specifics

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

Related Information

The **m_get** kernel service, **m_free** kernel service, **m_freem** kernel service.

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

m_clgetx Kernel Service

Purpose

Allocates an **mbuf** structure whose data is owned by someone else.

Syntax

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

struct mbuf *m_clgetx (*fun, arg, addr, len, wait***)**
int (**fun***)( );**
int *arg*;
caddr_t *addr*;
int *len*;
int *wait*;

Parameters

fun	Identifies the address of a function to be called when the **mbuf** structure is freed.
arg	Specifies an argument to pass to the function specified by the *fun* parameter.
addr	Specifies the address of the external data area.
len	Specifies the length of the external data area.
wait	This flag indicates the action to be taken if no **mbuf** structures are available. Possible values are:

M_DONTWAIT	Called from either an interrupt or process environment
M_WAIT	Called from a process environment only. The **M_WAIT** flag does not return until the **mbuf** structure is available.

Description

The **m_clgetx** kernel service allocates an **mbuf** structure and attaches the data specified by the *addr* parameter. This data is owned by the caller. The **m_off** field of the returned **mbuf** structure points to the caller's data.

Execution Environment

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

Return Values

The **m_clgetx** 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_clgetx** kernel service returns NULL.

Implementation Specifics

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

Related Information

The **m_clget** kernel service.

I/O Kernel Services in *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

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

Related Information

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

m_copy 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_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** 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. If the *len* parameter is set to M_COPYALL, the entire **mbuf** chain is copied.

Execution Environment

The **m_copy** kernel service 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

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

Related Information

The **m_copydata** kernel service.

I/O Kernel Services in *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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **m_copy** kernel service.

I/O Kernel Services in *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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **mbreq** Structure for **mbuf** Kernel Services.

The **m_reg** kernel service.

I/O Kernel Services in *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 value of NULL is returned if the structure specified by the *m* parameter is not part of an **mbuf** chain.

Related Information

The **m_get** kernel service.

I/O Kernel Services in *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_free (*m*)
**struct mbuf ****m*;

Parameter

m	Indicates the head of the **mbuf** chain to be freed.

Description

The **m_freem** kernel service invokes the **m_free** kernel service for each **mbuf** structure in the chain headed by the head specified by the *m* parameter. The **m_freem** service has no return values.

Execution Environment

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

Implementation Specifics

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

Related Information

The **m_get** kernel service, **m_free** kernel service.

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

m_get Kernel Service

Purpose

Allocates a memory buffer 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 **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 only specify the *wait* parameter as M_DONTWAIT.

Return Value

The address of an allocated **mbuf** structure is returned upon successful completion. If the *wait* parameter is set to M_DONTWAIT and there are no free **mbuf** structures, the **m_get** kernel service returns NULL.

Related Information

The **m_free** kernel service, **m_freem** kernel service.

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

m_getclr Kernel Service

Purpose

Allocates and zeros 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 **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, 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.

Return Values

The **m_getclr** kernel service returns the address of an allocated **mbuf** structure. If the *wait* parameter is set to M_DONTWAIT and there are no free **mbufs**, the **m_getclr** kernel service returns a NULL value.

Related Information

The **m_free** kernel service, **m_freem** kernel service, **m_get** kernel service.

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

m_getclust Kernel Service

Purpose

Allocates an **mbuf** structure from the **mbuf** buffer pool and attaches a page-sized cluster.

Syntax

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

struct mbuf *m_getclust(*wait, type***)**
int *wait*;
int *type*;

Parameters

wait	This flag 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 **sys/mbuf.h** file.

Description

The **m_getclust** kernel service allocates an **mbuf** structure of the specified type. If the allocation succeeds, the **m_getclust** kernel service 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** kernel service does not return until an **mbuf** structure is available.

Execution Environment

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

Return Value

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** kernel service returns a value of NULL.

Related Information

The **m_get** kernel service, **m_clget** kernel service, **m_free** kernel service, **m_freem** kernel service.

I/O Kernel Services in *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.

The **M_HASCL** macro can be used as in the following example:

Example

1. The **M_HASCL** macro can be used as in the following example:

```
struct mbuf     *m;
if (M_HASCL(m))
        printf("mbuf has attached cluster");
```

Execution Environment

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

Implementation Specifics

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

Related Information

I/O Kernel Services in *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 the data area of the **mbuf** structure 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

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

Related Information

I/O Kernel Services in *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.

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

Execution Environment

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

Related Information

The **mbreq** Structure for **mbuf** Kernel Services.

The **m_dereg** kernel service.

I/O Kernel Services in *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.

Format

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

  struct mbreq {
      int   low_mbuf;              /* mbuf low-water mark
*/
      int   low_clust;  /* page–sized mbuf low-water mark
*/
      int   initial_mbuf;          /* initial allocation of mbufs
*/
      int   initial_clust;         /* initial allocation of page-sized mbufs
*/
  }
```

Description

The **mbreq** structure specifies the **mbuf** structure usage expectations for a user of **mbuf** kernel services.

Implementation Specifics

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

Related Information

The **m_reg** kernel service, **m_dereg** kernel service.

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

mbstat Structure for mbuf Kernel Services

Purpose

Contains **mbuf** usage statistics.

Format

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

struct mbstat {
ulong   m_mbufs;          /* number of mbufs allocated */
ulong   m_clusters;       /* number of clusters allocated */
ulong   m_space;          /* number of interface pages */
ulong   m_clfree;         /* number of free clusters */
ulong   m_drops;          /* times failed to find space */
ulong   m_wait;           /* times waited for space */
ulong   m_drain;          /* times drained protocols for space */
short   m_mtypes[256];    /* type–specific mbuf allocations */
}
```

Description

The **mbstat** structure provides usage information for the **mbuf** services. Statistics can be viewed through the **–m** option of the **netstat** command.

Related Information

The **netstat** command.

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

mincnt Routine for the uphysio Kernel Service

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 *mincnt* parameter can be modified by that routine:

- **b_bcount**
- **b_work**
- **b_options**.

No other fields can be modified by the *mincnt* routine without the risk of error. If the *mincnt* routine 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 strategy 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* routine.

Implementation Specifics

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

Related Information

The **uphysio** kernel service.

I/O Kernel Services, Processing by the uphysio Kernel Service in *Kernel Extensions and Device Support Programming Concepts.*

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

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

Related Information

M_HASCL macro for **mbuf** Kernel Services.

I/O Kernel Services in *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/types.h>
#include <sys/errno.h>
#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:

Example

1. The **MTOD** 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.

Related Information

DTOM macro for **mbuf** Kernel Services.

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

net_attach Kernel Service

Purpose

Opens an AIX 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	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, the value 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

This kernel service is part of AIX 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 *Kernel Extensions and Device Support Programming Concepts.*

net_detach Kernel Service

Purpose

Closes an AIX 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 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 Value

The **net_detach** service returns the value it obtains from the **fp_close** service.

Implementation Specifics

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

Related Information

The **fp_close** kernel service, **net_attach** kernel service.

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

net_error Kernel Service

Purpose

Handles errors for AIX 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

ifp	Specifies the address of the **ifnet** structure for the device with an error.
error_code	Specifies the error code listed in the **comio.h** header file.
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 are of the following types:

- Errors received from the communications adapter drivers
- Errors occurring during input and output packet processing.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **net_attach** kernel service, **net_detach** kernel service.

Network Kernel Services in *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

1	Indicates that the sleeper was awakened by a signal.
0	Indicates that the sleeping process was not awakened by a signal.

Implementation Specifics

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

Related Information

The **net_wakeup** kernel service, **sleep** kernel service.

Understanding Locking, Network Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

net_start Kernel Service

Purpose

Starts network IDs on an AIX 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** operation 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

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

Related Information

The **net_start_done** kernel service, **net_attach** kernel service, **fp_ioctl** kernel service.

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

net_start_done Kernel Service

Purpose

Starts the done notification handler for AIX 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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **net_attach** kernel service, **net_start** kernel service.

Network Kernel Services in *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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **net_sleep** kernel service, **wakeup** kernel service.

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

net_xmit Kernel Service

Purpose

Transmits data using an AIX 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.

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.
Values returned from the fp_rwuio service	Indicates that an error occurred during the call to the **fp_rwuio** service.

Implementation Specifics

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

Related Information

The **fp_rwuio** kernel service.

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

notify Routine for the fp_select Kernel Service

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.

The entry point for the **notify** routine must be declared as follows:

#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

The parameters for the **notify** routine must be the following:

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* (channel for multiplexed drivers; 0 (zero) for nonmultiplexed drivers) parameter.
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.

Execution Environment

The **fp_select** kernel service **notify** routine can be called from either the process or interrupt environment.

Implementation Specifics

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

Related Information

The **fp_select** kernel service, **selnotify** kernel service.

Logical File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

RAS Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

peekq Kernel Service

Purpose

Returns a pending queue element in the device queue.

Syntax

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

struct req_qe *peekq (*queue_id, offset***)**
cba_id *queue_id*;
int *offset*;

Parameters

queue_id	Specifies the queue identifier.
offset	Specifies the offset of the queue element from the top of the device queue. The offset must be in the range from 1 to QE_MAX_OFFSET.

Description

The **peekq** 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 once before the loading of any kernel extensions referencing these services.

A device queue server can use the **peekq** kernel service to browse the contents of any but the active queue element in a device queue. This service does not remove the queue element from the device queue when it browses. Use the **readq** service to browse the active queue element in a device queue.

An offset value of 1 returns the most favored pending queue element.

The queue element includes the path identifier used to send the queue element. This value can be used with the **queryp** service to determine the process that sent the queue element.

Although most device queues process queue elements in first-in-first-out (FIFO) order, some device queues do not. Note that, for device queues not processed in FIFO order, the order of the queue elements can change after a queue element has been read at a particular offset. A device queue that does not process queue elements in FIFO order is one that has multiple priorities.

Pending queue elements can be deleted from the queue at any time. When using the **peekq** service to find elements on which requests are to be initiated, the caller must define a **cancel-queue-element** routine (with the **creatq** service) for his device queue.

Warning: The server must not alter any fields in the queue element or the system may crash.

Execution Environment

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

Return Values

Upon successful completion, the **peekq** service returns the address of the pending queue element. A value of NULL is returned when the queue identifier (specified by the *queue_id* parameter) is invalid or when there is no queue element at the specified offset.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **readq** kernel service, **queryp** kernel service, **creatq** kernel service.

The **cancel-queue-element** queue management routine.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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 on 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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

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

Understanding Socket Header Files in *Communications 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 (zero) 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

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

Related Information

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

Understanding Socket Header Files in *Communications Programming Concepts* .

pgsignal Kernel Service

Purpose

Sends a signal to a process group.

Syntax

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

void pidsignal (*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 process 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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **pidsig** kernel service, **getpid** kernel service.

Understanding Signal and Exception Handling.

The **sigaction** subroutine.

Understanding Signal and Exception Handling, Process and Exception Management Kernel Services in *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	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.

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.

See the **sigaction** subroutine for a list of the valid signals.

The **pidsig** kernel service can be called from an interrupt handler execution environment if the process ID is known. The **pidsig** service has no return values.

Execution Environment

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

Implementation Specifics

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

Related Information

The **pgsignal** kernel service, **getpid** kernel service.

The **sigaction** subroutine.

Understanding Signal and Exception Handling.

Understanding Signal and Exception Handling, Process and Exception Management Kernel Services in *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 *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 *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.
ENOMEM	Indicates that the **pin** service was unable to pin due to insufficient real memory or exceeding the systemwide pin count.

Implementation Specifics

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

Related Information

The **unpin** kernel service.

Understanding Execution Environments, Memory Kernel Services in *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** kernel service manages two lists of free character buffers. One list, which contains free unpinned character buffers, is used to allocate a character buffer when executing under a process. This allocation typically occurs during output operations.

The second list contains free pinned character buffers and is used to allocate a character buffer when executing on an interrupt level. This typically happens during input operations.

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.

A device driver calls the **pincf** service, typically during its **ddopen** routine, to tell the kernel how many character buffers the device driver intends to allocate from an interrupt level. A device driver also frees allocated **clist** resources by calling the **pincf** service with a *delta* parameter having a negative value. This happens typically during the device driver's **ddclose** routine.

Execution Environment

The **pincf** kernel service can be called in the process environment if the *delta* parameter has a positive value.

It can be called in either the process or interrupt environment if the *delta* parameter has a negative value.

Return Value

The **pincf** service returns a value representing the amount by which the service changed the number of free pinned character buffers.

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

Implementation Specifics

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

pincode Kernel Service

Purpose

Pins the code and data associated with an object file.

Syntax

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

int pincode (*func***)**
int (**func***) ();**

Parameter

func	Specifies the function in the object file to be pinned.

Description

The **pincode** service uses the **pin** service to pin the specified object file. The loader entry for the object file 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 due to insufficient real memory.

When an error occurs, the **pincode** service returns without pinning any pages.

Implementation Specifics

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

Related Information

The **pin** kernel service.

Understanding Execution Environments, Memory Kernel Services in *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 **<sys/uio.h>** header 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 **pinu** service is used to pin pages backing a specified memory region that 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.

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 *length* parameter is negative. 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 from exceeding the systemwide pin count.

Implementation Specifics

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

Related Information

The **unpinu** kernel service, **pin** kernel service.

Understanding Execution Environments, Memory Kernel Services in *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 where the use of the **pio_assist** service is contra-indicated due to efficiency and structure considerations.

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, and optionally 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 as 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 NULL, the *iofunc* routine is recalled to recover from I/O exceptions. This re-use of the I/O routine for error retry should only be used if the I/O routine can handle being recalled in the case of an error, 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 *ioparms* 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 sent over again without concern for hardware state synchronization problems.

Upon return from the I/O, the return code should be 0 (zero) 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 originally 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 was 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 is defined in the **<sys/except.h>** header 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 return code of EXCEPT_NOT_HANDLED 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 no errors are indicated by the *iofunc* I/O routine, 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 successful completion: either no errors were encountered, or PIO errors were encountered and successfully handled.
EIO	Indicates the failure of the IO operation: the maximum number of IO retry operations was exceeded.

Implementation Specifics

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

Related Information

Device Handler Error Recovery, Handling User-Mode Exceptions, Kernel-Mode Exception Handling, Kernel Extension/Device Driver Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

Process State-Change Notification Routine

Purpose

Allows kernel extensions to be notified of major process state transitions.

Syntax

The notification routine is called by the kernel as follows:

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 **<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	Specifies the process ID of the process.

Description

For process initialization, the 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 default termination procedures are handled by the kernel. 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 swapout when a process is about to be swapped out. At the point of call to the notification routine, the process's **u** block has not yet been unpinned.

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

The **prochadd** kernel service, **prochdel** kernel service.

Kernel Program/Device Driver Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

prochadd Kernel Service

Purpose

Adds a systemwide process state-change notification routine.

Syntax

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

void prochadd (*term***)**
**struct proch ****term*;

Parameter

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 about to be swapped out
- Has just been swapped in.

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 that is 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

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

Related Information

The **prochdel** kernel service.

Kernel Program/Device Driver Management Kernel Services in *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*;**

Parameters

term	Points to a **proch** structure containing a notification routine to be removed from the chain of systemwide 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 systemwide 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

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

Related Information

The **prochadd** kernel service.

Kernel Program/Device Driver Management Kernel Services in *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 as STALE) the specified block from the buffer cache. The **purblk** service has no return values.

Execution Environment

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

Implementation Specifics

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

Related Information

The **geteblk** kernel service, **brelse** kernel service.

Block I/O Buffer Cache Services: Overview, I/O Kernel Services in *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	Address of the **clist** structure that describes the character list.

Description

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, then the **waitcfree** service can be used to wait until a character block is available.

Warning: 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.

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

This kernel service is part of AIX 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 *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

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.

Warning: 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.

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

This kernel service is part of AIX 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 *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

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 next returns the number of characters added to the character list. If the character list is full and no more buffers are available, then the **putcbp** service returns a 0. Otherwise, it returns the number of characters written.

Warning: 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.

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

This kernel service is part of AIX 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 *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. The **putcf** service has no return values.

Execution Environment

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

Implementation Specifics

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

Related Information

I/O Kernel Services in *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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

I/O Kernel Services in *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 <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, then 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

This kernel service is part of AIX 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 *Kernel Extensions and Device Support Programming Concepts.*

qryds Kernel Service

Purpose

Returns information about the device associated with a device queue.

Syntax

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

int qryds (*device_id*, *ptr*, *count***)**
cba_id *device_id*;
caddr_t *ptr*;
int *count*;

Parameters

device_id	Specifies the device identifier.
ptr	Specifies the address of the buffer into which the device-dependent information is to be copied.
count	Specifies the size of the buffer in which to place the device-dependent information.

Description

The **qryds** 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 once before the loading of any kernel extensions referencing these services.

The **qryds** service returns the device information for the specified device. The service simply copies the data from the buffer (specified with the **creatd** service) into the caller's buffer.

A kernel extension should not return this data directly to the user application. It should simply extract selected fields as required and return only those fields. The kernel extension is responsible for ensuring that the information that it returns to the user does not constitute a security exposure.

The caller can specify a buffer size of 0 (indicated by the *count* parameter) to query only the size of the device-dependent information for the device (device-dependent information is not copied to the buffer).

Note: The **qryds** service is not serialized with respect to updates to the device-dependent information by the device manager. (That is, this is not an atomic operation.) Therefore, results may be unreliable.

Execution Environment

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

Return Values

The **qryds** service returns the size of the device-dependent information for the device, if successful. It returns a value of RC_NONE if an error was encountered accessing the device-dependent information or caller's buffer. The caller can use this to determine how much of the device-dependent information was returned.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **creatd** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

queryd Kernel Service

Purpose

Returns the device identifier associated with the specified IODN.

Syntax

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

cba_id queryd (*iodn*)
ushort *iodn*;

Parameter

iodn	Predetermined global name for a device queue.

Description

The **queryd** 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 once before the loading of any kernel extensions referencing these services.

The **queryd** service returns the device identifier that corresponds to thep redetermined global name specified by the *iodn* parameter. The name specified by the *iodn* parameter is associated with a device identifier and a device queue by the **creatd** service.

Execution Environment

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

Return Values

Upon successful completion, the **queryd** service returns the device identifier. A value of **NULL_CBA** is returned if the name specified by the *iodn* parameter is not currently assigned to a device queue.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **creatd** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

queryi Kernel Service

Purpose

Provides information about device queues.

Syntax

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

int queryi (*query_id*, *queue_ids*, *size*)
cba_id *query_id*;
struct queryi *queue_ids*;
int *size*;

Parameters

query_id	Specifies the identifier associated with a device queue.
queue_ids	Specifies the address of the array in which the device queue information is returned.
size	Specifes the number of elements in the array specifies by the *queue_ids* parameter.

Description

The **queryi** 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 once before the loading of any kernel extensions referencing these services.

The **queryi** service returns information about the device queues associated with the *query_id* parameter. The *query_id* parameter can specify a process identifier, a queue identifier, or a device identifier. The information returned is the device queue's identifier and event mask.

The *queue_ids* parameter is an array because a process can serve more than one device queue at a time.

Execution Environment

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

Return Values

The **queryi** service returns the number of entries inserted into the *queue_ids* array.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

queryp Kernel Service

Purpose

Indicates whether a path exists to a device queue.

Syntax

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

int queryp (*ptr*)
struct queryp **ptr*;

Parameter

ptr	Specifies the address of the query path structure.

Description

The **queryp** 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 once before the loading of any kernel extensions referencing these services.

The **queryp** service determines whether a path established by the **attchq** service exists and returns information about the path.

There are two ways to use the **queryp** service:

- To query the path identifier, the from-identifier and to-identifier must be filled in and the path identifier set to NULL_CBA. The from-identifier and the to-identifier can be a queue identifier, a device identifier, or a process identifier.
- To query the from-identifier and to-identifier, the path identifier must be filled in and the from-identifier and to-identifier set to NULL_CBA.

Other returned information concerning the path includes the acknowledgment type information and the server queue identifier. The **attchq** kernel service provides more details about these parameters.

Execution Environment

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

Return Values

RC_GOOD	Indicates successful completion.
RC_OBJ	Indicates that the path was not found. This can occur if the path does not exist or any of the input identifiers are invalid.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **attchq** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *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 **<net/raw_cb.h>** header file) and passes both on to the raw protocol input handler.

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

Execution Environment

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

Implementation Specifics

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

Related Information

Network Kernel Services in *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*, *rights*)
struct socket **so*;
int *req*;
struct mbuf **m*;
struct mbuf **nam*;
struct mbuf **rights*;

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.
rights	This parameter should be set to NULL.

Description

The **raw_usrreq** kernel service implements user requests for the raw protocol.

The **raw_usrreq** service supports the following commands:

PRU_ATTACH	PRU_DETACH
PRU_CONNECT	PRU_PEERADDR
PRU_BIND	PRU_DISCONNECT
PRU_SHUTDOWN	PRU_SEND
PRU_ABORT	PRU_SENSE
PRU_SOCKADDR	
PRU_CONTROL	PRU_CONNECT2
PRU_RCVOOB	PRU_RCVD
PRU_LISTEN	PRU_ACCEPT
PRU_SENDOOB	

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 for PRU_ATTACH.
ENOTCONN	Indicates an attempt to detach when not attached.
EISCONN	Indicates that the caller tried to connect while already connected.

Implementation Specifics

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

Related Information

The **panic** kernel service.

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

readq Kernel Service

Purpose

Returns the active queue element in the device queue.

Syntax

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

struct req_qe *readq (*queue_id*)
cba_id *queue_id*;

Parameter

queue_id	Specifies the device queue identifier.

Description

The **readq** 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 once before the loading of any kernel extensions referencing these services.

The **readq** kernel service returns the address of the active queue element in the device queue without actually removing the element from the queue. If there is currently no active queue element, the **readq** service makes the most favored queue element the active one.

Subsequent **readq** calls by the server return the same address until that queue element is removed from the device queue. Any queue elements placed on the device queue after the call to the **readq** service must wait for the active queue element to be removed from the device queue before being processed. This is true even if these queue elements have a more favored priority.

The path identifier in the queue element can be used as input to the **queryp** service to determine who sent the queue element.

Warning: The server must not alter any fields in the queue element or the system may crash.

Execution Environment

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

Return Values

Upon successful completion, the **readq** service returns the address of the active queue element in the device queue. A value of NULL is returned when the device queue is empty.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **queryp** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

rqc Kernel Service

Purpose

Creates a ring queue in the kernel heap.

Syntax

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

caddr_t rqc (*depth*, *event*, *pid*, *func*)
int *depth*;
unsigned long *event*;
unsigned int *pid*;
void(**func* **)();**

Parameters

depth	Indicates the requested number of elements in the queue.
event	Indicates the event mask.
pid	Specifies the ID of the process that will get data from the queue.
func	Points to a function to be used for notifying the process when the queue becomes non-empty. If this pointer is NULL, the process is to be posted.

Description

The **rqc** kernel service, along with the other ring queue services, 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 once before the loading of any kernel extensions referencing these services.

The **rqc** service creates a ring queue in the kernel heap. The ring queue contains ring queue private data followed by an array of elements of type **caddr_t**.

If the *func* parameter has a value of NULL, the *event* parameter is interpreted as an event mask and the process is posted. If the *func* parameter is non-NULL, it points to a special function that handles acknowledging that the queue is no longer empty. To differentiate between an empty and full queue, an extra, unused element is necessary. As a result, the *depth* parameter should be one more than actually needed.

Execution Environment

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

Return Values

The **rqc** service returns the address of the newly created ring queue or NULL if there is insufficient memory.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

Understanding Device Queues, Understanding Ring Queue Kernel Services, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

rqd Kernel Service

Purpose

Deletes a ring queue from the kernel heap.

Syntax

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

void rqd (*rqpointer*)
caddr_t *rqpointer*;

Parameter

rqpointer	Specifies the ring queue to be deleted.

Description

The **rqd** kernel service, along with the other ring queue services, 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 once before the loading of any kernel extensions referencing these services.

The **rqd** kernel service deletes a ring queue that was previously created with the **rqc** kernel service. The **rqd** service has no return values.

Execution Environment

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

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **rqc** kernel service.

Understanding Device Queues, Understanding Ring Queue Kernel Services, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

rqgetw Kernel Service

Purpose

Returns the next element from the specified queue.

Syntax

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

caddr_t rqgetw (*rqpointer***)**
caddr_t *rqpointer*;

Parameter

rqpointer	Specifies the ring queue from which the next element is to be returned.

Description

The **rqgetw** kernel service, along with the other ring queue services, 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 once before the loading of any kernel extensions referencing these services.

The **rqgetw** service gets the next element from the specified queue.

Execution Environment

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

Return Values

The **rqgetw** service returns the oldest element (of type **caddr_t**) in the queue, or NULL if the ring queue was empty.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

Understanding Device Queues, Understanding Ring Queue Kernel Services, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

rqputw Kernel Service

Purpose

Puts the specified element on the specified ring queue.

Syntax

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

int rqputw (*rqpointer*, *data***)**
caddr_t *rqpointer*;
caddr_t *data*;

Parameters

rqpointer	Specifies the ring queue that is to receive the new element.
data	Indicates the data to place in the queue. (This cannot be a value of –1.)

Description

The **rqputw** kernel service, along with the other ring queue services, 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 once before the loading of any kernel extensions referencing these services.

The **rqputw** service puts the specified element on the specified ring queue. If the ring queue was empty, the waiting process is notified that data is now available. If the ring queue was created with a non-NULL function pointer, then the notification is sent by calling the function. Otherwise, the notification is a pasting of the event that was set up.

Execution Environment

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

Return Values

0	Indicates successful completion.
–1	Indicates that the ring queue is full.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **rqc** kernel service.

Understanding Device Queues, Understanding Ring Queue Kernel Services, Device Queue Management Kernel Services in *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.

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

Execution Environment

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

Example

1. To allocate a route, invoke the **rtalloc** kernel service as follows:

```
rtalloc(ro);
```

Implementation Specifics

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

Related Information

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

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.

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

Execution Environment

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

Example

1. To free a routing table entry, invoke the **rtfree** kernel service as follows:

```
rtfree(rt);
```

Implementation Specifics

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

Related Information

The **panic** kernel service.

Network Kernel Services in *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 (*dst*, *gateway*, *cmd*, *flags*)
struct sockaddr **dst*, **gateway*
int *cmd*, *flags*;

Parameters

dst	Specifies the destination address.
gateway	Identifies the gateway address.
cmd	Specifies a request to add or delete route entry.
flags	Identifies routing flags, as defined in the **<net/route.h>** header file.

Description

The **rtinit** kernel service creates a routing table entry for an interface. It builds an **rtentry** structure using the values in the *dst*, *gateway*, and *flags* parameters.

The **rtinit** service then calls the **rtrequest** kernel service, passing it the *cmd* parameter and the **rtentry** structure, to process the request. The *cmd* parameter contains either the value SIOCADDRT (a request to add the route entry) or the value SIOCDELRT (delete the route entry). Valid routing flags to set are defined in the **<net.route.h>** header file.

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

Execution Environment

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

Example

1. To set up a routing table entry, invoke the **rtinit** kernel service as follows:

```
rtinit(dst, gateway, (int)SIOCADDRT, flags | RTF_DYNAMIC)
```

Implementation Specifics

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

Related Information

The **rtrequest** kernel service.

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

rtredirect Kernel Service

Purpose

Forces a routing table entry with the specified destination to go through the given gateway.

Syntax

#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>
#include <net/route.h>

rtredirect (*dst*, *gateway*, *flags*, *src*)
struct sockaddr **dst*, **gateway*, **src*;
int *flags*;

Parameters

dst	Specifies the destination address.
gateway	Specifies the gateway address.
flags	Routing flags as defined in the **<net/route.h>** header file.
src	Identifies the source of the redirect request.

Description

The **rtredirect** kernel service forces a routing table entry for the 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 **rtinit** kernel service to process the request.

Execution Environment

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

Example

1. To force a routing table entry with the specified destination to go through the given gateway, invoke the **rtredirect** kernel service:

```
rtredirect(dst, gateway, flags, src);
```

Return Values

0	Indicates a successful operation.

If a bad redirect request is received, the routing statistics counter for bad redirects is incremented.

Implementation Specifics

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

Related Information

The **rtinit** kernel service.

Network Kernel Services in *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*, *entry***)**
int *req*;
**register struct rtentry ****entry*;

Parameters

req	Specifies a request to add or delete a route.
entry	Specifies the routing table entry.

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 the value SIOCADDRT) or delete (the *req* parameter is SIOCDELRT) the route specified by the *entry* parameter.

Execution Environment

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

Example

1. To carry out a request to change the routing table, invoke the **rtrequest** kernel service as follows:

```
rtrequest(cmd, &route);
```

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.

Implementation Specifics

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

Related Information

The **rtinit** kernel service.

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

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. Interrupt service routines are established by the **add_netisr** kernel service. If the service was added with a service level of NET_OFF_LEVEL, the interrupt service routine is called directly from the **schednetisr** kernel service. If the service level was NET_KPROC, a network kernel process 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

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

Related Information

The **add_netisr** kernel service, **del_netisr** kernel service.

Network Kernel Services in *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	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	This subidentification parameter is used in conjunction with the primary resource identifier, *id*, to identify to the kernel the resource on which the event has occurred. For a multiplexed device driver, this is the number of the channel on which the requested events occurred. If the device driver is nonmultiplexed, then the *subid* parameter must be set to 0 (zero).
rtnevents	Returned events parameter. This parameter 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

Use of the selnotify Kernel Service

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 reawakened 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 was false.
- Asynchronous notification of the events was 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 *rtnevents,* the returned events parameter indicated by the call to the **selnotify** service, is no longer being waited on, no processes are awakened.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **ddselect** device driver entry point.

The **fp_select** kernel service, **fp_poll** kernel service.

The **select** subroutine, **poll** subroutine.

Select and Poll Support in Understanding Character I/O Device Drivers, Kernel Extension/Device Driver Management Kernel Services in *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

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

Related Information

The **longjmpx** kernel service, **clrjmpx** kernel service.

Stacking Saved Contexts for Nested setjmpx Calls, Exception Processing, Implementing Exception Handlers, Process and Exception Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

setpinit Kernel Service

Purpose

Sets the parent of the current kernel process to the init 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

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

Related Information

Using Kernel Processes, Process and Exception Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

setuerror Kernel Service

Purpose

Allows kernel extensions to set the **u_error** field in the **u** area.

Syntax

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

void setuerror (*errno***)**
int *errno*;

Parameter

errno	Contains a value found in the **sys/errno.h** header file that is to be copied to the current process's **u_error** field.

Description

The **setuerror** kernel service allows a kernel extension in a process environment to set the **u_error** field in the current process's **u** area. Kernel extensions providing system calls available to user-mode applications typically use this service. For system calls, the value of the **u_error** field in the per process **u** area is copied to the global variable **errno** by the system call handler before returning to the caller.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **getuerror** kernel service.

Actions of the System Call Handler describes the steps that the system call handler takes when a system call is invoked in user mode.

Returning Error Information describes how system calls return error information.

Understanding System Call Execution, Returning Error Information from System Calls, Kernel Program/Device Driver Management Kernel Services in *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

The **sig_chk** kernel service can be called by a kernel process to determine if any unmasked signals have been received. Signals do not preempt kernel processes because serialization of critical data areas would be lost. Instead, kernel processes 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 code. The signal is also removed from the pending signal mask. If no signal is found, this service will return a value of 0 (zero). Signals that are blocked or ignored will not be returned by this service.

The kernel does not take default action for signals delivered to kernel processes, but instead treats them as *caught* signals. The **sig_chk** service returns both *caught* and default signals. It is the responsibility of the kernel process to handle the signal appropriately. Unlike user processes, kernel processes do not have signal handlers automatically invoked by the kernel in response to *caught* signals.

Programming Kernel Processes provides details on the actions that the kernel takes when a signal is generated and delivered to a kernel process.

Warning: A system crash will occur if the **sig_chk** service is called by other than a kernel process.

Execution Environment

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

Return Value

Upon completion, the **sig_chk** service returns a return code of 0 (zero) if no pending unmasked signal is found. Otherwise, a nonzero signal value is returned indicating the number of the highest priority signal that is pending. Signal values are defined in the **<sys/signal.h>** file.

Implementation Specifics

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

Related Information

Introduction to Kernel Processes, Kernel Process Signal and Exception Handling, Process and Exception Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

sleep Kernel Service

Purpose

Forces the calling process 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 process 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 **<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**, **e_sleepl**, or **e_wait** kernel service should be used when writing new code.

The **sleep** service puts the calling process to sleep, causing it to wait for a wakeup to be issued for the channel specified by the *chan* parameter. When the process is awakened 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 processes. Thus, after returning from the **sleep** service, each process 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 *chan* value to start up all the processes that are 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, then 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.

Warning: 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

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

Related Information

Locking Strategy in Kernel Mode, Understanding Signal and Exception Handling, Process and Exception Management Kernel Services in *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 had 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

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

Related Information

The **suword** kernel service, **fubyte** kernel service, **fuword** kernel service.

Accessing User-Mode Data While in Kernel Mode, Memory Kernel Services in *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 EPERM 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 (zero)).

Note: The **suser** kernel service is supported for compatibility only and should not be called by new code.

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 (that is, the process has the specified privilege).

Implementation Specifics

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

Related Information

Security Kernel Services in *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

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

Related Information

The **subyte** kernel service, **fubyte** kernel service, **fuword** kernel service.

Accessing User-Mode Data While in Kernel Mode, Memory Kernel Services in *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. It must be called by the user 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

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

Related Information

The **tstart** kernel service, **tstop** kernel service, **tfree** kernel service.

Timer and Time-of-Day Kernel Services, Using Timer Services and Structures in *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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **tstart** kernel service, **tstop** kernel service, **talloc** kernel service.

Timer and Time-of-Day Kernel Services, Using Timer Services and Structures in *Kernel Extensions and Device Support Programming Concepts.*

timeout Kernel Service

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.
	The function specified by the *func* parameter should be declared as follows:
	`void func (arg)`
	`void *arg;`

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.

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 be pinned and be link-edited with the **libsys.a** library. This service and the associated **timeoutcf** subroutine and **untimeout** service disable interrupts for serialization and therefore must be included in the pinned part of the kernel extension or in the bottom half of the device driver.

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. The HZ label found in the **<sys/param.h>** file can be used to determine the number of ticks per second

Multiple pending **timeout** requests with the same *func* and *arg* parameters are not allowed.

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

Execution Environment

The **timeout** kernel service 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.

Implementation Specifics

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

Related Information

The **untimeout** kernel service.

The **timeoutcf** kernel subroutine.

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

timeoutcf Kernel Subroutine

Purpose

Allocates or deallocates callout table entries for use with the **timeout** kernel service.

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 be pinned and be link-edited with the **libsys.a** library. This subroutine and the associated **untimeout** and **timeout** kernel services disable interrupts for serialization and therefore must be included in the pinned part of the kernel extension or in the bottom half of the device driver.

The **timeoutcf** subroutine registers an increase or decrease in the number of callout table entries available for the **timeout** service to use. Before a subroutine can use the **timeout** service, the **timeoutcf** subroutine must increase the number of callout table entries available to the **timeout** 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** service 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** service for that instance of the open device.

Warning: A kernel panic results under either of these 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** kernel service 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

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

Related Information

The **timeout** kernel service.

Timer and Time-of-Day Kernel Services in *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 0x0EF.
	hk_type A 4-bit hook type. The **trcgenk** service automatically records this information.
data_word	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.

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** service simply returns. The **trcgenk** kernel service is located in pinned kernel memory.

The **trcgenk** 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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **trace** command.

The **trcstart** subroutine, **trcstop** subroutine, **trcon** subroutine, **trcoff** subroutine, **trchk** subroutine, **trcgen** subroutine, **trcgent** subroutine.

RAS Kernel Services in *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 0x0EF.
	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.	

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 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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **trace** command.

The **trcgenk** kernel service.

The **trcstart** subroutine, **trcstop** subroutine, **trcon** subroutine, **trcoff** subroutine, **trchk** subroutine, **trcgen** subroutine, **trcgent** subroutine.

RAS Kernel Services in *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** service must first call the **talloc** kernel service to allocate the timer request structure. It must then initialize the structure's fields before calling the **tstart** service.

Once the request has been submitted, the kernel calls the *t*->**func** timer function when the amount of time specified by *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** 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 **<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 1200 (noon) and a user submits an incremental timer request for 6 hours from now (to occur at 6:00 pm). If, before the timer expires, the privileged user modifies the system time by adding four hours to it, the user's timer request then expires at 2200 (10:00 pm).

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **talloc** kernel service, **tstop** kernel service, **tfree** kernel service.

Timer and Time-of-Day Kernel Services, Using Timer Services and Structures in *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>

void 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** service must be called before a timer request block can be freed with the **tfree** kernel service.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **tstart** kernel service, **talloc** kernel service, **tfree** kernel service.

Timer and Time-of-Day Kernel Services, Using Timer Services and Structures in *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 **<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** 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.

Warning: The **uexcepth** structure must be pinned when the **uexadd** service is called. It must remain pinned and unmodified until after the call to the **uexdel** service to delete the specified exception handler. Otherwise, the system may crash.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **uexdel** kernel service.

The User-Mode Exception Handler for the **uexadd** Kernel Service.

User-Mode Exception Handling Overview, Kernel Extension/Device Driver Management Services in *Kernel Extensions and Device Support Programming Concepts.*

uexblock Kernel Service

Purpose

Makes the currently active process non-runnable when called from a user-mode exception handler.

Syntax

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

void uexblock (*pid***)**
**pid_t ***pid*;

Parameter

pid	Specifies the process ID of the currently active process to be put into a wait state.

Description

Warning: The system will crash if the **uexblock** service is called in an interrupt handler environment or in a process environment that is not the process to be blocked.

The **uexblock** kernel service puts a currently active process specified by the *pid* parameter into a wait state until the **uexclear** kernel service is used to make the process runnable again.

The **uexblock** service can be used to lazily control user-mode process access to a shared serially usable resource. A serially used resource is usable by more than one process, but only by one at a time. When a process attempts to access the resource but does not have access, 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 process using the **uexblock** service until the resource is made available. At this time, the **uexclear** kernel service can be used to make the blocked process runnable.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **uexclear** kernel service.

User-Mode Exception Handling Overview, Kernel Extension/Device Driver Management Services in *Kernel Extensions and Device Support Programming Concepts.*

uexclear Kernel Service

Purpose

Makes a process blocked by the **uexblock** service runnable again.

Syntax

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

void uexclear (*pid***)**
**pid_t ***pid*;

Parameter

pid	Specifies the process ID of the process to be put into a runnable state.

Description

The **uexclear** kernel service puts a process specified by the *pid* parameter back into a runnable state after it was made nonrunnable by the **uexblock** kernel service. A process that has been sent a **SIGSTOP** stop signal is made runnable again when it receives the **SIGCONT** continuation signal.

The **uexclear** service can be used to lazily control user-mode process access to a shared serially usable resource. A serially used resource is usable by more than one process, but only by one at a time. When a process 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 process until the resource is later made available. At that time, the **uexclear** service can be used to make the blocked process runnable.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **uexblock** kernel service.

User-Mode Exception Handling Overview, Kernel Extension/Device Driver Management Services in *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** 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** service has completed, the specified exception handler is no longer called. In addition, the **uexcepth** structure can be modified, freed, or unpinned.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **uexadd** kernel service.

User-Mode Exception Handling Overview, Kernel Extension/Device Driver Management Services in *Kernel Extensions and Device Support Programming Concepts.*

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 (zero) 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 (zero), 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.
ENOMEM	Indicates that 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 is not valid.
–1	Indicates that an error occurred for one of the following conditions:

- 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 accessing the user buffer.

Implementation Specifics

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

Related Information

The **ureadc** kernel service, **uwritec** kernel service, **uphysio** kernel service.

Moving Large Numbers of Characters at a Time, The uio Structure, Memory Kernel Services in *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

The **unlockl** kernel service unlocks a conventional lock. Only the owner of a lock can unlock it. Once a lock is unlocked, all processes waiting for the lock are made runnable and allowed to recompete 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 the **unlockl** service 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.

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

Example

1. 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 before using resource */
        lock_ret = lockl(lock_word, LOCK_SHORT)
        if (lock_ret == LOCK_FAIL)
        {
            /* handle lock error */
            ....
        }
        else
        {
            /* use resource for which lock was obtained */
            ...
            /* release lock if this was not a nested use */
            if ( lock_ret |= LOCK_NEST )
                 unlockl(lock_word);
        }
```

Execution Environment

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

Implementation Specifics

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

Related Information

The **lockl** kernel service.

Preempting a System Call, Understanding Locking, Process and Exception Management Kernel Services in *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 (zero), 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 *length* parameter is negative. 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.

Related Information

The **pin** kernel service, **pinu** kernel service, **unpinu** kernel service.

Understanding Execution Environments, Memory Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

unpincode Kernel Service

Purpose

Unpins the code and data associated with an object file.

Syntax

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

int unpincode (*func***)**
int (**func***) ( );**

Parameter

func	Specifies the function in the object file to be unpinned.

Description

The **unpincode** kernel service uses the **unpin** kernel service to decrement the pin count for the pages associated with the following items:

- Code associated with the object file
- Data area of the object file 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.
EINVAL	Indicates that one or more pages are not pinned.

Implementation Specifics

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

Related Information

The **unpin** kernel service.

Understanding Execution Environments, Memory Kernel Services in *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 **<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 (zero), 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 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

Process environment when unpinning data that is in either user space or system space.

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* value.
EINVAL	Indicates that the *length* parameter is negative. 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

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

Related Information

The **unpin** kernel service, **pinu** kernel service, **pin** kernel service.

Understanding Execution Environments, Memory Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

untimeout Kernel Service

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	Function associated with the timer to be canceled.
arg	Function argument associated with the timer to be canceled.

Description

The **untimeout** 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 be pinned and be link-edited with the **libsys.a** library. This service and the associated **timeoutcf** subroutine and **timeout** service disable interrupts for serialization and therefore must be included in the pinned part of the kernel extension or in the bottom half of the device driver.

The **untimeout** service cancels a specific request made with the **timeout** service. The *func* and *arg* parameters must match those used in the **timeout** 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.

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

Execution Environment

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

Implementation Specifics

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

Related Information

The **timeout** kernel service.

Timer and Time-of-Day Kernel Services in *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 9.
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 invoked. 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* routine.

Description

Introduction

The **uphysio** 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** 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* routine is passed a pointer to the **buf** header along with the *minparms* pointer. This arrangement allows the *mincnt* routine to tailor the length of the data transfer described by the **buf** header as required by the device performing the I/O. The *mincnt* routine can also optionally modify certain device-dependent fields in the **buf** header.

When the *mincnt* routine 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* routine 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* routine indicates an error.
- The data area size is reduced to 0 (zero).

When insufficient memory indicates a failed pin operation, the number of **buf** headers used for the remainder of the operation is reduced to 1 (one). 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 **<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. A cross-memory descriptor is also obtained to allow the device driver interrupt handler some access to the data area.

Calling the Strategy Routine

This **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 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

In the case of error detection, the **uphysio** service reports the error that was detected for the **uio** structure closest to the start of the data area described by the **uio** structure. Once an error is detected by the **uphysio** service, no additional **buf** headers are sent to the strategy routine. The **uphysio** 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 the following two points in time:

- 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 this. 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 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* routine if the routine returned with a nonzero return code.

Implementation Specifics

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

Related Information

The **ddstrategy** device driver entry point.

The **iodone** kernel service, **geterror** kernel service.

The **mincnt** routine.

Processing by the uphysio Kernel Service, Block I/O Processing, The uio Structure, The buf Structure, Understanding Block I/O Device Drivers, I/O Kernel Services in *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 halves, especially character device drivers, frequently use the **ureadc** 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 (zero). 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 **<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 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

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

Related Information

The **uwritec** kernel service, **uiomove** kernel service, **uphysio** kernel service.

Reading One Character at a Time, The uio Structure, Memory Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

User-Mode Exception Handler for the uexadd Kernel Service

Purpose

Handles exceptions that occur while a process is executing in user mode.

Syntax

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

int func (*exp*, *type*, *pid*, *mst*)
struct excepth **exp*;
int *type*;
pid_t *pid*;
struct mstsave **mst*;

Parameters

exp	Points to the **excepth** structure used to register this exception handler.
type	Denotes the type of exception that has occurred. This type value is platform-specific. Specific values are defined in the **<sys/except.h>** file.
pid	Specifies the process ID of the user process that was executing at the time of the exception.
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.

Description

The user-mode exception handler (*exp*–>**func**) is called for synchronous exceptions that are detected while a process is executing in user mode. The kernel exception handler saves exception information in the **mstsave** area. 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 in the list is the default handler. This is typically signalling the process.

The 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 I/O is being performed by the exception handler. If the exception was not handled by the exception handler, 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 AIX Base Operating System (BOS) Runtime.

Related Information

The **uexadd** kernel service.

User-Mode Exception Handling Overview, Kernel Extension/Device Driver Management Services in *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 halves, especially character device drivers, frequently use the **uwritec** 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 (zero) 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 **<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 occured while the location was being accessed.

Implementation Specifics

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

Related Information

The **ureadc** kernel service, **uiomove** kernel service, **uphysio** kernel service.

Writing One Character at a Time, Memory Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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** 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 once before the loading of any kernel extensions referencing these services.

The **vec_clear** service removes the association between a virtual interrupt handler and the virtual interrupt level and sublevel that was assigned by the **vec_init** service. The virtual interrupt handler at the sublevel specified by the *levsublev* parameter will no longer be registered 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** service has no return codes. If no virtual interrupt handler is registered at the specified sublevel, no operation is performed.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **vec_init** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

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** 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 once before the loading of any kernel extensions referencing these services.

The **vec_init** 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 level and assigned sublevel are then returned by the **vec_init** service.

There is a maximum number of available sublevels. If this number is exceeded, the **vec_init** service crashes 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** service returns a value that identifies the virtual interrupt level and sublevel assigned. The low-order 8 bits of this value specify the sublevel, and the high-order 8 bits specify the level. This is the same format used by the **attchq** service. This level value is the same value as that supplied by the *level* parameter.

Implementation Specifics

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **attchq** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

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_UNMOUNTING** flag is set in the **vfs** structure, indicating that it is no longer valid to do pathname-related operations within the file system. When this flag is set and a VN_RELE vnode operation releases the last active vnode within the file system, the VN_RELE vnode 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 Value

The **vfsrele** service always returns a value of 0 (zero).

Implementation Specifics

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

Related Information

Understanding Virtual Nodes (vnode), Virtual File System Overview, Virtual File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

virtual-interrupt-handler Queue Management Routine

Purpose

Provides a means for notifying requestors when a request has completed.

Syntax

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

void vintrh (*arg*, *qe***)**
int *arg*;
struct ack_qe **qe*;

Parameters

arg	Specifies a virtual interrupt handler argument provided on the call to **vec_init** kernel service to register the virtual interrupt handler.
qe	Specifies the address of the acknowledge queue element supplied on the call to the **ackque** or **deque** kernel service.

Description

The **virtual-interrupt-handler** routine is called to notify the requestor of the completion of a request. This routine is called when the **deque** or **ackque** service is called for a queue element that had a virtual interrupt acknowledgement specified. Understanding Device Queues describes device queue elements.

This routine is registered by calling the **vec_init** service, and is assigned an available sublevel. This sublevel can then be specified when creating a path to the queue using the **attchq** service when a virtual interrupt acknowledge is specified. A different sublevel can be specified in the queue element to override the one specified during an **attchq** call when using the **deque** service to dequeue the element and send the acknowledgement.

This routine is called from the **ackque** and **deque** services when a virtual interrupt acknowledgement has been specified. This routine runs in the process environment of the caller of the **ackque** and **deque** services. It is passed the *arg* parameter supplied when the virtual interrupt handler was initialized and a pointer to the acknowledge queue element provided on the call to the **ackque** and **deque** services.

Execution Environment

The **virtual-interrupt-handler** routine runs in the process environment of the server but is typically established by the requestor. Therefore access to data areas must be handled as if the routine is executed in the interrupt environment. However, it is not necessary to pin the data and code since the routine is executed in the process environment.

Related Information

The **ackque** kernel service, **deque** kernel service, **vec_init** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *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** service assumes an address space model of fixed-size virtual memory objects and address space regions.

Warning: If there are no more free regions, this call cannot complete and calls the **panic** service.

Execution Environment

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

Return Values

The **vm_att** service returns the address that corresponds to the *offset* parameter in the address space.

Implementation Specifics

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

Related Information

The **vm_det** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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 zero, 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 may have any alignment in a page.

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

Execution Environment

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

Implementation Specifics

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

Related Information

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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** service assumes an address space model of fixed-size virtual memory objects and address space regions.

Warning: If the region is not mapped, or a system region is referenced, a system crash occurs.

Execution Environment

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

Implementation Specifics

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

Related Information

The **vm_att** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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 0 value for limited access and a 1 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** service to map a virtual memory object into the current address space.

The **vm_handle** 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 Value

The **vm_handle** service returns a virtual memory handle type.

Implementation Specifics

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

Related Information

The **vms_create** kernel service, **vm_att** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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** 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 an invalid virtual memory object type or invalid page number.
EFBIG	Indicates that the page number exceeds the file-size limit.

Implementation Specifics

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

Related Information

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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

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

Related Information

The **vm_umount** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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 zero, no bytes are transferred.
rw	Specifies a read/write flag that gives the direction of the move. The possible values for this flag (UIO_READ, UIO_WRITE) are defined in the **sys/uio.h** header file.
uio	Points to the **uio** structure.

Description

The **vm_move** kernel service moves data between 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** service is similar to the **uiomove** 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** 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.

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.

Execution Environment

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

Implementation Specifics

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

Related Information

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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 five 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

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

Related Information

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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 (zero), 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

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

Related Information

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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	Specifes 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 zero, 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 four 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

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

Related Information

The **vm_releasep** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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	Specifes 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 non-negative.

Each page of the virtual memory object that intersects the page range (*pfirst*, *pfirst* + *npages* – 1) is logically reset to 0 (zero), 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 these four 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

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

Related Information

The **vm_release** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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

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

Related Information

The **vm_mount** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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 pageout 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	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 (zero). 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

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

Related Information

The **vms_iowait** kernel service, **vm_writep** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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 (zero).

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 Value

0	Indicates successful completion.
EINVAL	Indicates any one of these four errors:

- An invalid virtual memory object ID.
- The starting page is negative.
- The number of pages is negative.
- The page range exceeds the size of virtual memory object.

Implementation Specifics

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

Related Information

The **vm_write** kernel service, **vms_iowait** kernel service.

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

vms_create Kernel Service

Purpose

Creates a virtual memory object of the type and size and limits specified.

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* field 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 gnode for client storage.
size	Specifies the current size of the file. This can be any valid file size.
uplim	This parameter is ignored. The enforcement of file size limits is done by comparing with the **u_limit** value in the **u** block.
downlim	This parameter is 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

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

Related Information

The **vms_delete** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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 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

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

Related Information

The **vms_create** kernel service.

Memory Kernel Services, Understanding Virtual Memory Manager Interfaces in *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

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

Related Information

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

vn_free Kernel Service

Purpose

Frees a vnode 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 vnode to be deallocated.

Description

The **vn_free** kernel service provides the only acceptable mechanism for deallocation of vnode objects used within the AIX virtual file system. The vnode specified by the *vp* parameter is removed from the linked list of vnodes belonging to the owning virtual file system. The vnode is then returned to the free pool of vnodes for re-use.

Execution Environment

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

Return Value

The **vn_free** service always returns 0 (zero).

Implementation Specifics

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

Related Information

The **vn_get** kernel service.

Virtual File System Overview, Virtual File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

vn_get Kernel Service

Purpose

Allocates a virtual node and inserts it into the list of vnodes for the designated virtual file system.

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 vnode. Any returned vnode belongs to this virtual file system.
gnp	Points to the gnode for the object. This pointer is stored in the returned vnode. The new vnode is added to the list of vnodes in the gnode.
vpp	Points to the place in which to return the vnode pointer. This is set by the **vn_get** service to point to the newly allocated vnode.

Description

The **vn_get** service provides the only acceptable mechanism for allocating vnode objects for use within the AIX virtual file system environment. A vnode is first allocated from an effectively infinite pool of available vnodes. It is then inserted on the linked list of vnodes currently active for the virtual file system specified by the *vfsp* parameter.

Upon successful return from the **vn_get** service, the pointer to the vnode pointer provided (specified by the *vpp* parameter) has been set to the address of the newly allocated vnode.

The fields in this vnode have been initialized as follows:

v_count	Set to 1 (one).
v_vfsp	Set to the value in the *vfsp* parameter.
v_gnode	Set to the value in the *gnp* parameter.
v_vfsnext	Specifies a forward pointer to a vnode that belongs to the same virtual file system.
v_vfsprev	Specifies a backward pointer to a vnode that belongs to the same virtual file system.

All other fields in the vnode 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** service could not allocate memory for the vnode. (This is a highly unlikely occurrence.)

Implementation Specifics

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

Related Information

The **vn_free** kernel service.

Virtual File System Overview, Virtual File System Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

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>

void 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** 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.

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

Execution Environment

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

Implementation Specifics

This kernel service is part of AIX 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, The Watchdog Timer Function in *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>

void w_init (*w***)**
struct watchdog ***w*;**

Parameter

w	Specifies the watchdog timer structure.

Description

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.

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

Warning: 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.

Execution Environment

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

Implementation Specifics

This kernel service is part of AIX 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, The Watchdog Timer Function in *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** 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 (zero). A watchdog timer is ignored when the *w*–>**count** watchdog count is less than or equal to 0.

The **w_start** service sets the *w*–>**count** watchdog count to a value of *w*–>**restart**.

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

Warning: The watchdog structure must be pinned when the **w_start** 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.

Execution Environment

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

Implementation Specifics

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

Related Information

The **w_init** kernel service, **w_clear** kernel service, **w_stop** kernel service.

Timer and Time-of-Day Kernel Services, The Watchdog Timer Function in *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** 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 (zero). A watchdog timer is ignored when *w*–>**count** is less than or equal to 0.

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

Warning: The watchdog structure must be pinned when the **w_stop** 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.

Execution Environment

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

Implementation Specifics

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

Related Information

The **w_init** kernel service, **w_clear** kernel service, **w_start** kernel service.

Timer and Time-of-Day Kernel Services, The Watchdog Timer Function in *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 checks whether there is an available free character buffer. If one is not available, the **waitcfree** service waits until either a character buffer becomes available or a signal is received.

The **waitcfree** 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

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

Related Information

The **putc** kernel service, **putcb** kernel service, **putcbp** kernel service, **putcx** kernel service, **putcf** kernel service, **putcfl** kernel service.

I/O Kernel Services in *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 provided by the Device Queue Management kernel extension. This queue management kernel extension must be loaded into the kernel once before the loading of any kernel extensions referencing these services.

The **waitq** 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 only:

- Waits on the event mask associated with the device queue
- Calls the **readq** service to make the most favored queue element the active one.

Processes can only use the **waitq** service to wait for a single device queue. Use the **e_wait** service to wait on the occurrence of more than one event, such as multiple device queues.

The **waitq** service uses the EVENT_SHORT form of the **e_wait** service. Therefore a signal does not terminate the wait. Use the **e_wait** service if you want a signal to terminate the wait.

The **readq** service can be used to read the active queue element from a queue. It does not wait for a queue element if there is none in the queue.

Warning: The server must not alter any fields in the queue element or the system may crash.

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

This kernel service is part of the Device Queue Management AIX kernel extension.

Related Information

The **readq** kernel service, **e_wait** kernel service.

Understanding Device Queues, Device Queue Management Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

wakeup Kernel Service

Purpose

Activates processes sleeping on the specified channel.

Syntax

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

void wakeup (*chan*)
int *chan*;

Parameter

chan	Specifies the channel number. For the **wakeup** service, this parameter identifies the event that the processes are waiting for.

Description

Note: The **wakeup** kernel service is provided for compatibility only. Either the **e_wakeup** or **e_post** service should be used instead.

The **wakeup** kernel service readies suspended processes for execution. It affects all processes that were suspended by using the **sleep** service on the channel specified by the *chan* parameter. The processes do not begin to execute until the current process relinquishes control of the processor or returns to user mode.

A race condition occurs when all processes that are waiting on the channel are restarted. Thus, after returning from the **sleep** service, each process should check to see whether it needs to sleep again.

Note: The possible existence of a race condition indicates that all processes whose sleep channel hashes to the same value are awakened together, not just those with the equivalent channel values.

The channel specified by the *chan* parameter is a value that identifies an event to wait for or to sleep on. This value is passed to the **wakeup** service to start all of the processes that are waiting for the event. The channel identifier must be unique on a systemwide basis. As a result, the address of an external kernel variable (which can be defined in a kernel extension) is generally used for this value.

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

Execution Environment

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

Implementation Specifics

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

Related Information

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

Watchdog Timer Function

Description

The watchdog timer services support the watchdog timer function. The *w*–>**func** watchdog timer function should be declared as follows:

void *func* (*w*)
struct watchdog **w*;

The kernel calls the watchdog timer function when the watchdog timer is decremented to 0 (zero).

Execution Environment

The **watchdog** timer function can be called from either the process or interrupt environment.

Implementation Specifics

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

Related Information

The **w_init** kernel service, **w_clear** kernel service, **w_start** kernel service, **w_stop** kernel service.

Timer and Time-of-Day Kernel Services, The Watchdog Timer Function in *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 **kernel_heap** when allocating memory that is not pinned. They should also use **pinned_heap** when allocating memory that is not pinned. When allocating memory that is to be always pinned (or pinned for long periods of time), the **pinned_heap** should be specified. The memory is pinned upon successful return from the **xmalloc** service. When allocating memory that can be pageable (or only pinned for short periods of time), the **kernel_heap** 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** service (**128**, **3**, **kernel_heap**) allocates a 128–byte double word aligned area out of the kernel heap.

A kernel extension must use the **xmfree** service to free the allocated memory. If it does not, subsequent allocations eventually fail.

xmalloc Compatibility Interfaces: malloc and palloc

The following additional interfaces to the **xmalloc** kernel service are provided:

- **malloc** (*size*) is equivalent to **xmalloc** (*size*, **3**, **kernel_heap**).
- **palloc** (*size*, *align*) is equivalent to **xmalloc** (*size*, *align*, **kernel_heap**).

Execution Environment

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

The **xmalloc** routines are part of AIX Base Operating System (BOS) Runtime.

Return Values

Upon successful completion, the **xmalloc** kernel service returns the address of the allocated area. A NULL pointer is returned under the following two circumstances:

- The requested memory cannot be allocated.
- The heap has not been initialized for memory allocation.

Implementation Specifics

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

Related Information

The **xmfree** kernel service.

Memory Kernel Services in *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	Cross-memory descriptor. The *dp*->**aspace_id** variable must be set to a value of XMEM_INVAL.
segflag	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 **xmem.h** header 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** service. The **xmattach** service allows a kernel process or device bottom-half routine to access the user buffer with the **xmemin** or **xmemout** services. The device driver must use the **xmdetach** 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. Using Cross-Memory Kernel Services describes how the cross-memory kernel services use cross-memory descriptors.

Note: When the **xmattach** 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 these four errors:

- The buffer crosses a segment boundary.
- The buffer size indicated by the *count* parameter is less than or equal to 0 (zero).
- 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

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

Related Information

The **uphysio** kernel service, **xmdetach** kernel service, **xmemin** kernel service, **xmemout** kernel service.

Using Cross-Memory Kernel Services, Memory Kernel Services in *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** 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 re-used. Using Cross-Memory Kernel Services 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 invalid or the buffer was not defined.

Implementation Specifics

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

Related Information

The **xmemin** kernel service, **xmemout** kernel service, **xmattach** kernel service.

Using Cross-Memory Kernel Services, Memory Kernel Services in *Kernel Extensions and Device Support Programming Concepts.*

xmemdma Kernel Service

Purpose

Prepares a page for 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_HIDE	Prepare the page for DMA I/O. This hides the page by making it not accessible.
XMEM_UNHIDE	Process the page after DMA I/O. This unhides the page, which is the default.

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** 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** service instead of the **xmemdma** service.

If the *flag* parameter has an XMEM_HIDE value and this is the first hide for the page, the page is prepared for DMA I/O by flushing the cache and making the page not valid. When the *flag* parameter has the value XMEMUNHIDE and this is the last unhide for the page, the following three 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.

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.

XMEMFAIL	Indicates that the descriptor was not valid, or the page specified by the *xaddr* or *dp* parameter is invalid, or the page was not pinned.

Implementation Specifics

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

Related Information

Direct Memory Access (DMA).

Using Cross-Memory Kernel Services, Direct Memory Access (DMA), Memory Kernel Services in *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** service copies data from the specified address space to kernel global memory.

The **xmemin** service is provided so that kernel processes and interrupt handlers can safely access a buffer within a user process. Calling the **xmattach** service prepares the user buffer for the cross-memory move.

The **xmemin** service differs from the **copyin** and **copyout** services in that it is used to access a user buffer when not executing under the user process. In contrast, the **copyin** and **copyout** services are only used 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 these five errors:

- The user does not have the appropriate access authority for the user buffer.
- The user buffer is located in an invalid address range.

- The segment containing the user buffer has been deleted.
- The cross-memory descriptor is invalid.
- A paging I/O error occurred while the user buffer was being accessed.

If the user buffer is not in memory, the **xmemin** service also returns an XMEM_FAIL error code when executing on an interrupt level.

Implementation Specifics

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

Related Information

The **xmattach** kernel service, **xmdetach** kernel service, **xmemout** kernel service.

Using Cross-Memory Kernel Services, Memory Kernel Services in *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

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 **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** service copies data from kernel global memory to the specified address space.

The **xmemout** service is provided so that kernel processes and interrupt handlers can safely access a buffer within a user process. Calling the **xmattach** service prepares the user buffer for the cross-memory move.

The **xmemout** service differs from the **copyin** and **copyout** services in that it is used to access a user buffer when not executing under the user process. In contrast, the **copyin** and **copyout** 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 the process environment only.

Return Values

XMEM_SUCC Indicates successful completion.

XMEM_FAIL Indicates one of these five errors:

- The user does not have the appropriate access authority for the user buffer.
- The user buffer is located in an invalid address range.

- The segment containing the user buffer has been deleted.
- The cross-memory descriptor is invalid.
- A paging I/O error occurred while the user buffer was being accessed.

If the user buffer is not in memory, the **xmattach** service also returns an XMEM_FAIL error code when executing on an interrupt level.

Implementation Specifics

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

Related Information

The **xmattach** kernel service, **xmdetach** kernel service, **xmemin** kernel service.

Using Cross-Memory Kernel Services, Memory Kernel Services in *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** service. In addition, the *ptr* pointer must be the pointer returned from the corresponding **xmalloc** call.

For example, the **xmfree** (*ptr*, **kernel_heap**) 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, subsequent allocations eventually fail. 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.

xmfree Compatibility Interface: free

The following additional interface to the **xmfree** kernel service is provided:

- **free** (*ptr*) is equivalent to **xmfree** (*ptr*, **kernel_heap**).

Execution Environment

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

The **free** routine is part of AIX Base Operating System (BOS) Runtime.

Return Values

0 Indicates successful completion.

–1 Indicates one of these two errors:

- The area to be freed was not allocated with the **xmalloc** service.
- The heap was not initialized for memory allocation.

Implementation Specifics

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

Related Information

The **xmalloc** kernel service.

xmfree Compatibility Interface: **free**.

Memory Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

Chapter 2. Device Driver Operations

Guide to Writing Device Driver Entry Points

The following articles are provided as guidance for programming the major routines of a device driver.

Prerequisite Information

Introduction to Block and Character Device Driver Entry Points

Parameters Common to Most Device Driver Entry Points.

Structures for Device Driver Entry Points

The **buf** Structure

Character Lists Structure

Device Dependent Structure (DDS)

The **uio** Structure.

Requirements for Individual Device Driver Entry Points

The **ddconfig** Device Driver Entry Point

The **ddopen** Device Driver Entry Point

The **ddclose** Device Driver Entry Point

The **ddioctl** Device Driver Entry Point

The **dddump** Device Driver Entry Point

The **ddread** Device Driver Entry Point

The **ddwrite** Device Driver Entry Point

The **ddselect** Device Driver Entry Point

The **ddmpx** Device Driver Entry Point

The **ddrevoke** Device Driver Entry Point

The **ddstrategy** Device Driver Entry Point.

Character and Block Device Driver Entry Points: Overview

Introduction

Each device driver providing a device head role is invoked by the kernel using standard entry points, also called interface routines. Each major device number has a corresponding set of entry points named **ddconfig**, **ddopen**, **ddclose**, **ddioctl**, **ddread**, **ddwrite**, **ddstrategy**, **ddselect**, **ddmpx**, **ddrevoke**, and **dddump**. By convention, the prefix **dd** uniquely identifies a particular device driver.

Device Driver Requirements for Individual Entry Points

All device drivers require the **ddconfig**, **ddopen** and **ddclose** entry points.

Both block and character class device drivers can have a **ddioctl** entry point to provide special control functions. Character device drivers and block device drivers providing *raw I/O* access to their devices can have **ddread** and **ddwrite** routines.

Only the block class of device driver can have a **ddstrategy** routine for performing block I/O.

Only the character class of device driver can have a **ddselect** routine for notifying applications of requested events while the **ddmpx** routine is only provided by multiplexed character device drivers for allocating and de-allocating channels.

The **ddrevoke** routine needs to be provided only by device drivers in the Trusted Computing Path for a user in order to terminate processes currently waiting in the device driver. The **dddump** entry point is provided by device drivers when their respective devices can be selected as an output device for system dump data.

Functions of the Individual Entry Points

Entry Points Common to Character and Block Class Device Drivers

ddconfig	Called typically during the system start-up procedures to configure or unconfigure the device driver. Also called during runtime when devices are added or removed from the system.
ddopen	Called when the device is opened with an **open** or **creat** subroutine call to get the device ready to transfer data. This entry point provides access to a device instance for its calling program. Where necessary, this access can be exclusive to the opener.
ddclose	Called when the device is to be closed. Puts the device in a known idle state. This entry point removes access to a device instance.
ddioctl	Called when a user program invokes the **ioctl** subroutine. Decodes commands for special functions.
dddump	Called to write system dump data to the device. This must be serviced by a bottom-half routine.

Entry Points for Character Device Drivers and Raw Access Block Drivers

ddread	Called when the user program issues a **read** subroutine call to a character device.
ddwrite	Called when the user program issues a character device **write** subroutine.

Entry Points for Character Device Drivers Only

ddselect	Called when the user program issues a **select** or **poll** subroutine call to a character device.
ddmpx	Required for multiplexed device drivers to provide allocation and deallocation of a channel.

Entry Points for Block Device Drivers Only

ddstrategy	Called to schedule a read or write to a block device. Performs block data transfer to or from the device.

Entry Points for Trusted Computing Path Device Drivers

ddrevoke	Provided by character device drivers in the Trusted Computing Path to a user's terminal. This routine is typically invoked when the Secure Attention Key (**SAK**) is detected to insure that a secure path to the user's terminal is being provided.

Using Examples of Device Driver Source Code

Warning: These device driver source code examples are only intended to assist in the development of a working software program. These examples do not function as written. Additional code is required.

Related Information

The **close** subroutine, **ioctl** subroutine, **lseek** subroutine, **open** subroutine, **poll** subroutine, **read** subroutine, **select** subroutine, **write** subroutine.

Device Driver Classes, Device Driver Roles, Device Driver Structure in *Kernel Extensions and Device Support Programming Concepts.*

Multiplexed Support For Character Device Drivers, Trusted Computing Path Support For Character Device Drivers in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Block I/O Device Drivers, Understanding Character I/O Device Driver, Understanding Major and Minor Numbers For A Special File, Understanding Raw I/O Access to Block Devices in *Kernel Extensions and Device Support Programming Concepts.*

Standard Parameters to Device Driver Entry Points

Three of the parameters passed to device driver entry points always have the same meanings. These three parameters are described here.

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 **<sys/sysmacros.h>** header 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 (zero). If the driver is multiplexed, then *chan* is the **chan_t** value returned from the device driver's **ddmpx** routine.

The ext Parameter

The *ext* parameter, which is the extension parameter, is defined to be of type **int** and 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 **read**, instead of **readx**), 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.

File

sys/sysmacros.h

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 Drivers Kernel Extension Overview, Programming in the Kernel Environment, Understanding I/O Access to Special Files, Understanding Major and Minor Numbers For a Special File in *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 the following 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 **<sys/buf.h>** header file, includes the following fields:

b_flags Flag bits. The value of this field is constructed by logically OR-ing 0 (zero) 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_WANTED	The **e_wakeup** kernel service should be called when the block is released.
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_STALE	The data conflicts with the data on disk because of an I/O error.
B_REMOTE	This block is associated with a remote file.

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.

Cross Memory Kernel Services, Device Driver Classes, Device Driver Roles, Processing Interrupts, Providing Raw I/O Access in a Block Device Driver, Providing Raw I/O Support, Understanding Block I/O Device Drivers, Understanding Interrupts, Understanding Major and Minor Numbers for a Special File in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *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 know 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. Registration for character block usage is provided by the **pincf** kernel service.

The kernel provides four services for obtaining characters or character blocks from a character list: the **getc**, **getcb**, **getcbp**, and **getcx** services. Four services are also provided that add characters or character blocks to character lists: the **putc**, **putcb**, **putcbp**, and **putcx** services. The **getcf** services allocates a free character block while the **putcf** service returns a character block to the free list. Additionally, the **putcfl** service returns a list of character buffers to the free list. The **waitcfree** service determines if any character blocks are on the free list, and wait for one if none are available.

Using a clist

For each character list that you use, you must allocate a **clist** header structure. This **clist** structure is defined in the **cblock.h** header 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 (zero), and both character block pointers, **c_cf** and **c_cl**, to NULL before using the **clist** header for the first time. These fields are all defined by the **clist** structure.

Each buffer in the character list is a **cblock** structure, which is also defined in the **cblock.h** header file.

A character block data area does not need to be completely filled with characters. The fields **c_first** and **c_last** are zero-based offsets within the **c_data** array, which actually contains the data.

The amount of memory available for character buffers is limited. 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 that all of its character lists are flushed so that the buffers are returned to the list of free buffers.

File

cblock.h

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.

Character I/O Device Drivers, Device Driver Classes, Device Driver Roles, Device Drivers Kernel Extension Overview, Execution Environments, Programming In the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts*.

Device Dependent Structure (DDS)

Description

A *Device Dependent Structure* (DDS) contains information that describes a device instance to the device driver. It typically contains information about device-dependent attributes as well as other information the driver needs to communicate with the device. In many cases, information about a device's parent is included. For instance, a driver needs information about the adapter, and the bus the adapter is plugged into, to communicate with a device connected to an adapter.

A device's DDS is built each time the device is configured. The Configure method can fill in the DDS with fixed values, computed values, and information from the Configuration database. Most of the information from the Configuration database usually comes from the attributes for the device in the Customized Attribute (CuAt) object class, but can come from any of the object classes. Information from the database for the device's parent device or parent's parent device can also be included. The DDS is passed to the device driver with the SYS_CFGDD option of the **sysconfig** subroutine, which calls the device driver's **ddconfig** routine with the CFG_INIT command.

How the Change Method Updates the DDS

The Change method is invoked when changing the configuration of a device. The Change method must ensure consistency between the Configuration database and the view that any device driver may have of the device. This is accomplished by:

1. Not allowing the configuration to be changed if the device has configured children, that is, children in either the Available or Stopped states. This ensures that a DDS that has been built using information in the database about a parent device will remain valid because the parent cannot be changed.
2. If a device has a device driver and the device is in either the Available or Stopped states, the Change method must communicate to the device driver any changes that would affect the DDS. This may be accomplished with **ioctl** operations, if the device driver provides the support to do so. It can also be accomplished by taking the following steps:
   a. Terminating the device instance by calling **sysconfig** subroutine with the SYS_CFGDD option. The SYS_CFGDD operation calls the device driver's **ddconfig** routine with the CFG_TERM command.
   b. Rebuilding the DDS using the changed information.
   c. Passing the new DDS to the device driver by calling the **sysconfig** SYS_CFGDD operation. This operation then calls the **ddconfig** routine with the CFG_INIT command.

Many Change methods simply invoke the device's Unconfigure method, apply changes to the database, then invoke the device's Configure method. This process ensures the two stipulated conditions since the Unconfigure method, and thus the change, will fail, if the device has Available or Stopped children. Also, if the device has a device driver, its Unconfigure method terminates the device instance. Its Configure method also rebuilds the DDS and passes it to the driver.

Guidelines for DDS Structure

There is no single defined DDS format. Writers of device drivers and device methods must agree upon a particular device's DDS format. When obtaining information about a parent device, you may want to group that information together in the DDS.

When building a DDS for a device connected to an adapter card, you will typically need to pick up the following adapter information:

slot number Obtained from the **connwhere** descriptor of the adapter's Customized Device (CuDv) object.

bus resources Obtained from attributes for the adapter in the Customized Attribute (CuAt) or Predefined Attribute (PdAt) object classes. These include attributes for bus interrupt levels, interrupt priorities, bus memory addressed, bus I/O addresses, and DMA arbitration levels.

These two attributes must be obtained for the adapter's parent bus device:

bus_id Identifies the I/O bus. This field is needed by the device driver to access the I/O bus.

bus_type Identifies the type of bus such as a Micro Channel bus or a PC AT bus.

Note: The **getattr** device configuration subroutine should be used whenever attributes are obtained from the Configuration Database. This routine returns the Customized attribute value if the attribute is represented in the Customized Attribute object class. Otherwise, it returns the default value from the Predefined Attribute object class.

Finally, a DDS generally includes the device's logical name. This is used by the device driver to identify the device when logging an error for the device.

Example of DDS

```
/* Device DDS */
struct device_dds {
   /* Bus information */
   ulong   bus_id;       /* I/O bus id                         */
   ushort  bus_type;     /* Bus type, i.e. BUS_MICRO_CHANNEL */

   /* Adapter information */
   int     slot_num;            /* Slot number                */
   ulong   io_addr_base;        /* Base bus i/o address       */
   int     bus_intr_lvl;        /* bus interrupt level        */
   int     intr_priority;       /* System interrupt priority */
   int     dma_lvl;             /* DMA arbitration level      */

   /* Device specific information */
   int     block_size;          /* Size of block in bytes     */
   int     abc_attr;            /* The abc attribute          */
   int     xyz_attr;            /* The xyz attribute          */
   char    resource_name[16];   /* Device logical name        */
};
```

Related Information

The **ddconfig** device driver entry point.

The **getattr** subroutine, **ioctl** subroutine, **sysconfig** subroutine.

The SYS_CFGDD **sysconfig** operation.

ODM Device Configuration Object Classes, Writing a Change Method, Writing a Configure Method.

Basic Device Configuration Procedures Overview, Writing A Device Method in *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 **<sys/uio.h>** header 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.

Kernel services are provided 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** service writes a character into the buffer described by the **uio** structure. The **uwritec** service reads a character from the buffer. Thus, these two services have names opposite from what one 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** 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** service is used primarily by block device drivers providing raw I/O support. The **uphysio** 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 non-contiguous 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 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 following fields are contained in the **uio** structure:

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** 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 **<sys/uio.h>** header 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 **<sys/fcntl.h>** header 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.
iovec structure	A structure containing 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 may 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 number of **iovec** structures in the array is given by the **uio_iovcnt** field.
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.

Files

sys/uio.h

sys/fcntl.h

sys/xmem.h

Related Information

The **ddread** device driver entry point and **ddwrite** device driver entry point.

The **uiomove** kernel service, **uphysio** kernel service, **ureadc** kernel service, **uwritec** kernel service.

The **fcntl** subroutine, **lseek** subroutine.

Accessing Data from a Kernel Process, Accessing User-Mode Data While in Kernel Mode, The Cross Memory Kernel Services, Providing Raw I/O Access in a Block Device Driver, Understanding Block I/O Device Drivers in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, Programming In the Kernel Environment in *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 **ddclose** routine is called by the kernel when the last process having the device instance open closes it. This causes the gnode reference count to be decremented to 0 (zero), and the gnode to be de-allocated.

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 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 used by this device.

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 de-allocate 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 non-zero 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** external variable. The return code used should be one of the values defined in the **<sys/errno.h>** header 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 Classes, Device Driver Roles, Multiplexed Support in a Character Device Driver, Non-Multiplexed Support in a Character Device Driver, Processing Interrupts, Providing Raw I/O Access in a Block Device Driver, Providing Raw I/O Support, Understanding Block I/O Device Drivers, Understanding Character I/O Device Drivers, Understanding Direct Memory Access, Understanding Interrupts, Understanding Locking in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *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 and is determined by the specific device method and **ddconfig** routine.

Additional device-specific functions relating to configuration can also be provided by the **ddconfig** routine, such as returning device vital product data (VPD). The **ddconfig** routine is usually invoked through the **sysconfig** subroutine by the device-specific Configure method.

The values for the *cmd* parameter typically supported by device drivers and their methods are:

CFG_INIT	Initialize the device driver and internal data areas.
CFG_TERM	Terminate the device driver associated with the specified device number, *devno*.
CFG_QVPD	Query device-specific VPD.

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 may be used to facilitate the copying of information into or out of this data area. The format of the **uio** structure is defined in the **<sys/uio.h>** header 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 (zero) 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 **<sys/errno.h>** header 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** external variable and providing a –1 return code to the subroutine.

Files

sys/uio.h

sys/errno.h

Related Information

The **uiomove** kernel service.

The **sysconfig** subroutine.

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

CFG_INIT Command Parameter to the ddconfig Routine

Purpose

The **CFG_INIT** value is one of three common values for the *cmd* parameter to the **ddconfig** device driver entry point. The **CFG_INIT** value requests initialization of a particular device driver and device.

Description

This command type is used to specify that the **ddconfig** routine is to perform an initialization function, which typically involves checking the minor number in *devno* 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** code 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.

Execution Environment

This routine is called in the process environment only.

Related Information

The **CFG_QVPD** command parameter, **CFG_TERM** command parameter.

The **ddconfig** device driver entry point, **ddopen** device driver entry point.

The **uiomove** kernel service, **devswadd** kernel service.

The **uio** structure.

Understanding the Device Switch Table, Understanding Major and Minor Numbers for a Special File in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

CFG_QVPD Command Parameter to the ddconfig Routine

Purpose

The **CFG_QVPD** value is one of three common values for the *cmd* parameter to the **ddconfig** device driver entry point. The **CFG_QVPD** value queries for device-specific vital product data (VPD).

Description

The **CFG_QVPD** command is an optional **ddconfig** function called from the device's Configure method. It is usually used for diagnostic purposes.

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.

Execution Environment

This routine is called in the process environment only.

Related Information

The **CFG_INIT** command parameter, **CFG_TERM** command parameter.

The **ddconfig** device driver entry point.

The **uiomove** kernel service.

The **uio** structure.

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

CFG_TERM Command Parameter to the ddconfig Routine

Purpose

The **CFG_TERM** command option is one of three common values for the *cmd* parameter to the **ddconfig** device driver entry point. The **CFG_TERM** value requests termination of a particular device driver.

Description

The **CFG_TERM** command type is typically used by a device's Unconfigure or Change method through the **sysconfig** subroutine to remove resources and system access for a specific device. The **ddconfig** routine should determine if any opens are outstanding on the specified *devno*. If none are, the **CFG_TERM** command processing should mark 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*.

If opens are outstanding on the specified device, the terminate operation should be 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 may learn whether or not the device driver has removed itself from the device switch table.

Execution Environment

This routine is called in the process environment only.

Related Information

The **CFG_INIT** command parameter, **CFG_QVPD** command parameter.

The **ddconfig** device driver entry point.

The **devswdel** kernel service.

The SYS_QDVSW **sysconfig** operation.

The **sysconfig** subroutine.

Understanding the Device Switch Table, Understanding Major and Minor Numbers for a Special File in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *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 **dddump** entry point is called by the kernel dump routine 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 in preparation for supporting a system dump.
DUMPQUERY	Query minimum and maximum data transfer sizes.
DUMPSTART	Device setup in preparation for doing a system dump.
DUMPWRITE	Write dump data to the device.
DUMPEND	Cleanup of the device state after completing dump.
DUMPTERM	Release resources allocated for dump support.

Return Value

The **dddump** entry point indicates an error condition to the caller by returning a nonzero return code.

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.

Related Information

The **devdump** kernel service, **dmp_add** kernel service, **dmp_del** kernel service.

The **dump** special file.

Possible Values for the *Cmd* Parameter to the **dddump** Device Driver Entry Point.

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

Device Driver System Dump Support: Possible Values for the dddump cmd Parameter

Introduction

The *cmd* parameter to the **dddump** device driver entry point takes six possible values: **DUMPINIT**, **DUMPQUERY**, **DUMPSTART**, **DUMPWRITE**, **DUMPEND**, and **DUMPTERM**, as defined in the **<sys/devide.h>** header file.

The DUMPINIT cmd Value

The **DUMPINIT** command is sent when this device has been selected as the target dump device for the kernel. The specified device instance must have previously been opened.

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.

The DUMPQUERY cmd Value

The **DUMPQUERY** command is sent by the kernel dump function to determine the maximum and minimum number of bytes that can be transferred to the device in one **DUMPWRITE** command. For this command, the *uiop* parameter is not used and is NULL. The *arg* parameter is a pointer to a **dmp_query** structure, as defined in the **<sys/device.h>** header file. This structure contains the following fields:

min_tsize	Minimum transfer size (in bytes)
max_tsize	Maximum transfer size (in bytes).

The **DUMPQUERY** command returns the data transfer size information in the **dmp_query** structure pointed to by the *arg* parameter. The kernel dump function will then use 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 may specify the same value.

The DUMPSTART cmd Value

In response to the **DUMPSTART** command, the **dddump** routine must suspend current device activity and provide whatever setup of the device is needed before receiving a **DUMPWRITE** command.

The DUMPWRITE cmd Value

The **DUMPWRITE** command is sent to write 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. 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 will have been 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.

The DUMPEND cmd Value

The **DUMPEND** command is sent to indicate that the kernel dump has been completed. Any cleanup of the device state should be done at this time.

The DUMPTERM cmd Value

The **DUMPTERM** command is sent to indicate 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.

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.

Related Information

The **dddump** device driver entry point.

The **pin** kernel service, **pincode** kernel service, **unpin** kernel service.

The **uio** structure.

Understanding Interrupts, Processing Interrupts in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *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** 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 **<sys/devinfo.h>** header file) that describes the device. (Refer to the description of the special file for a particular device in the AIX 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 **<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 non-blocking 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** external variable. The error code used should be one of the values defined in the **<sys/errno.h>** header file.

When applicable, the return values defined in the POSIX 1003.1 standard for the **ioctl** subroutine should be used.

File

/sys/device.h

Related Information

The **fp_ioctl** kernel service.

The **fcntl** subroutine, **ioctl** subroutine, **ioctlx** subroutine, **open** subroutine.

Special Files Overview in *Files Reference*.

Device Drivers Kernel Extension Overview, File System Overview, Programming in the Kernel Environment, Understanding Locking in *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 pathname extension for the channel to be allocated.

Description

Only multiplexed character class device drivers may provide the **ddmpx** routine, and *every* multiplexed driver must do so. The **ddmpx** routine may not 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 pathname of the special file for the device. This pathname 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 **ddmpx** routine is called by the kernel when a channel is to be allocated or deallocated. Upon allocation, the kernel dynamically creates gnodes (in-core inodes) for channels on a multiplexed device to allow the protection attributes to be different for various channels.

To allocate a channel, the **ddmpx** routine is called with a *channame* pointer to the pathname extension. The pathname extension starts after the first / (slash) character that follows the special file name in the pathname. The **ddmpx** routine should perform the following actions:

- Parse this pathname extension.
- Allocate the corresponding channel.
- Return the channel ID through the *chanp* parameter.

If no pathname 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 gnode for the associated channel has its reference count incremented. If the channel was not already allocated, a new gnode 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 gnode reference count going to zero), 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 gnode reference count has gone to 0 (zero), 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.

Return Value

If the allocation or de-allocation of a channel is successful, the **ddmpx** routine should return a return code of 0. If an error occurs on allocation or de-allocation, a nonzero return code should be returned.

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 **<sys/errno.h>** header file.

Execution Environment

This routine is called in the process environment only.

Related Information

The **ddclose** device driver entry point, **ddopen** device driver entry point.

The **close** subroutine, **creat** subroutine, **open** subroutine.

Special Files Overview in *Files Reference*.

Device Driver Classes, Device Driver Roles, Multiplexed Support in a Character Device Driver, Non-Multiplexed Support in a Character Device Driver, Providing Raw I/O Access in a Block Device Driver, Providing Raw I/O Support, Understanding Block I/O Device Drivers, Understanding Character I/O Device Drivers, Understanding Generic Inodes (Gnodes), Understanding I/O Access to Device Drivers in *Kernel Extensions and Device Support Programming Concepts*.

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *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 ext*;
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 zero, the **ddopen** routine returns with a return code of **EBUSY** to indicate that the device is already open. Otherwise, the 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, and 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 open the device handler by using the **fp_opendev** kernel service.

   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 flag word *devflag* has the following flags, as defined in the **<sys/devide.h>** header 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 non-blocking 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** external variable. The return code used should be one of the values defined in the **errno.h** header 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 non–multiplexed 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.

File

/usr/include/errno.h

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 Classes, Device Driver Roles, Multiplexed Support in a Character Device Driver, Non-Multiplexed Support in a Character Device Driver, Processing Interrupts, Providing Raw I/O Access in a Block Device Driver, Providing Raw I/O Support, Understanding Block I/O Device Drivers, Understanding Character I/O Device Drivers, Understanding Direct Memory Access, Understanding Interrupts in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *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 point the read should start.

If no error occurs, the **uio_resid** field should be zero 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** external variable. The error code used should be one of the values defined in the **<sys/errno.h>** header 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 **fp_rwuio** kernel service, **e_sleep** kernel service, **e_wakeup** kernel service, **uphysio** kernel service, **ureadc** kernel service, **uiomove** kernel service.

Select/Poll Logic for the **ddread** Routine.

The **uio** structure.

The **read, readx** subroutines.

Interfacing to the Hardware, Processing Interrupts, Providing Raw I/O Access in a Block Device Driver, Providing Raw I/O Support, Understanding Block I/O Device Drivers, Understanding Character I/O Device Drivers, Understanding Interrupts, Understanding Locking in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

Select/Poll Logic for the ddread Routine

Description

The **ddread** entry point requires 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.

The **selnotify** kernel service.

The **poll** subroutine, **select** subroutine.

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

ddrevoke Device Driver Entry Point

Purpose

Ensures that a secure path to a terminal is being 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 **/dev/hft** and **/dev/tty** for the high 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 kill all processes waiting in the device driver while accessing the specified device or channel. It should kill 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 killed.

If the device driver is multiplexed and the channel ID in the *chan* parameter has the value –1, all channels are to be revoked.

The purpose of this function is to ensure that no "Trojan horses" exist in the Trusted Computing Path to the user.

Execution Environment

The **ddrevoke** routine is called in the process environment only.

Return Values

This routine should return a value of 0 for successful completion, or a value from the **errno.h** header file on error.

Files

/dev/hft

/dev/tty

sys/device.h

Related Information

The **frevoke** subroutine, **revoke** subroutine.

Device Driver Classes, Device Driver Roles, Multiplexed Support in a Character Device Driver, Providing Raw I/O in a Block Device Driver, Understanding Block I/O Device Drivers, Understanding Character I/O Device Drivers, Understanding Locking, Understanding Trusted Computing Path Support in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, HFT Subsystem Conceptual Introduction, Programming in the Kernel Environment, TTY Subsystem Overview in *Kernel Extensions and Device Support Programming Concepts.*

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.

The **ddselect** routine can be provided only by character class device drivers. It cannot be provided by block device drivers even when providing raw read/write access.

Requests for Information on Events

Possible events to check for are represented as flags (bits) in the *events* parameter. 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 asychronous notification of the requested events:

POLLSYNC	This request is a synchronous request only. The routine need not call the **selnotify** 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 in fact true, the **ddselect** routine should indicate this by setting the corresponding bits in the *reventp* parameter. Note that the returned events parameter *reventp* is passed by reference.

If none of the requested events are true, then the **ddselect** routine sets the returned events parameter to 0 (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 simply return, since the event request was a synchronous request only.

However, if the **POLLSYNC** flag is false, the **ddselect** routine needs to 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 renotification 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 (zero) if the select/poll operation requested is valid for the resource specified. Requested operations are invalid, however, if either of the following is true:

1. The device driver does not support a requested event.
2. 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 **select** subroutine, **poll** subroutine.

Device Driver Classes, Device Driver Roles, Providing Raw I/O in a Block Device Driver, Understanding Block I/O Device Drivers, Understanding Character I/O Device Drivers, Understanding Locking in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *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 above additions, the device's start I/O routine must be called.

Required Processing

The **ddstrategy** routine may receive a single request with multiple **buf** structures, and is required to process these blocks in sequential order. However, it is not required to process requests in their arrival 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** flag values.

Warning: 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.

Warning: 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 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 may 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.

The Buffer Cache kernel services, Device Driver Classes, Device Driver Structure, Understanding Block I/O Device Drivers in *Kernel Extensions and Device Support Programming Concepts*.

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *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** 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 which point the write should start.

If no error occurs, the **uio_resid** field should be zero 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 translates requests into block I/O requests using the **uphysio** kernel service.

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 Value

The **ddwrite** entry point can indicate an error condition to the caller by returning a nonzero return code. 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** external variable. The error code used should be one of the values defined in the **<sys/errno.h>** header 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.

Select/Poll Logic for the **ddwrite** Routine.

The **uio** structure.

The **write** subroutine, **writex** subroutine.

The **e_sleep** kernel service, **e_wakeup** kernel service, **fp_rwuio** kernel service, **uiomove** kernel service, **uphysio** kernel service, **uwritec** kernel service.

Interfacing to the Hardware, Processing Interrupts, Providing Raw I/O Access in a Block Device Driver, Providing Raw I/O Support, Understanding Block I/O Device Drivers, Understanding Character I/O Device Drivers, Understanding Interrupts, Understanding Locking in *Kernel Extensions and Device Support Programming Concepts.*

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

Select/Poll Logic for the ddwrite Routine

Description

The **ddwrite** entry point requires 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 **ddselect** device driver entry point, **ddwrite** device driver entry point.

The **selnotify** kernel service.

The **poll** subroutine, **select** subroutine.

Device Drivers Kernel Extension Overview, Programming in the Kernel Environment in *Kernel Extensions and Device Support Programming Concepts.*

Chapter 3. File System Operations

vfs_cntl Subroutine

Purpose

Issues control operations for a file system.

Syntax

int vfs_cntl (*vfsp*, *cmd*, *arg*, *argsize***)**
**struct vfs ****vfsp*;
int *cmd*;
caddr_t *arg*;
unsigned long *argsize*;

Parameters

vfsp	Points to the file system for which the control operation is to be issued.
cmds	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.

Description

The **vfs_cntl** subroutine issues control operations for a 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 functions 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 has no *argsize* parameter.

Execution Environment

The **vfs_cntl** subroutine can be called from the process environment only.

Return Values

0	Success.
EINVAL	The *cmd* parameter is not a supported control, or the *arg* parameter is an invalid argument for the command.
EACCESS	The *cmd* parameter requires a privilege that the current process does not have.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Related Information

The **fscntl** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes) in *Kernel Extensions and Device Support Programming Concepts.*

vfs_init Subroutine

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** subroutine initializes a virtual file system. It is called when a file system implementation is loaded to perform file system-specific initialization.

The **vfs_init** subroutine is not called through the virtual file system switch. Instead, it is called indirectly by the **gfsadd** kernel service when the **vfs_init** subroutine 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 **<sys/gfs.h>** file.

Note: The return value for the **vfs_init** subroutine is passed back as the return value from the **gfsadd** kernel service.

Execution Environment

The **vfs_init** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Related Information

The **gfsadd** kernel service.

File System Overview, List of Virtual File System Operations, Understanding Data Structures and Header Files for Virtual File Systems, The Virtual File System Kernel Services, Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vfs_mount Subroutine

Purpose

Mounts a virtual file system.

Syntax

int vfs_mount (*vfsp***)**
struct vfs ****vfsp*;**

Parameters

vfsp	Points to the newly created **vfs** structure.

Description

The **vfs_mount** subroutine mounts a virtual file system. This subroutine is called after the **vfs** structure is allocated and initialized. Before this structure is passed to the **vfs_mount** subroutine, the logical file system does the following:

- Guarantees the syntax of the **vmount** or **mount** subroutines.
- Allocates the **vfs** structure.
- Resolves the stub to a virtual node (vnode). This is the **vfs_mntdover** field in the **vfs** structure.
- Initializes the following virtual file system information:

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 stub. **VFS_SUSER** — The user has root user authority.
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 vnode 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	The unique number sequence representing this virtual file system.
vfs_mdata	This field is initialized with the **vmount** structure supplied by the user. The virtual file system data is detailed in the **<sys/vmount.h>** header file. All arguments indicated by this field are copied to kernel space.

Execution Environment

The **vfs_mount** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Files

/usr/include/sys/vfs.h	Defines the **vfs** structure.
/usr/include/sys/vmount.h	Defines types of virtual file systems.

Related Information

The **mount** subroutine, **vmount** subroutine.

File System Overview, Logical File System Overview, List of Virtual File System Operations, Understanding Data Structures and Header Files for Virtual File Systems, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vfs_root Subroutine

Purpose

Finds the root of a virtual file system (VFS).

Syntax

int vfs_root (*vfsp*, *vpp***)**
struct vfs ****vfsp*;**
struct vnode *****vpp*;**

Parameters

vfsp	Points to the **vfs** structure.
vpp	Points to the place to return the vnode pointer.

Description

The **vfs_root** subroutine finds the root of a virtual file system. When successful, the *vpp* parameter points to the root virtual node (vnode) and the vnode hold count is incremented.

Execution Environment

The **vfs_root** subroutine can be called from the process environment only.

Return Values

0	Success
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Files

/usr/include/sys/vfs.h	Defines the **vfs** structure.
/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

File System Overview, List of Virtual File System Operations, Understanding Data Structures and Header Files for Virtual File Systems, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vfs_statfs Subroutine

Purpose

Obtains virtual file system statistics.

Syntax

int vfs_stafs (*vfsp, statfsp***)**
**struct vfs ****vfsp*;
**struct statfs ****stafsp*;

Parameters

vfsp	Points to the **vfs** structure being queried. This structure is defined in the **<sys/vfs.h>** file.
stafsp	Points to a **statfs** structure. This structure is defined in the **<sys/statfs.h>** file.

Description

The **vfs_stafs** subroutine is used to return information specific to the virtual file system. When completed successfully, this subroutine fills in the following fields of the **statfs** structure:

f_blocks	Number of blocks.
f_files	Total number of file system objects.
f_bsize	File system block size.
f_bfree	Number of free blocks.
f_ffree	Number of free file system objects.
f_fname	A 32-byte string specifying the file system name.
f_fpack	A 32–byte string specifying a pack ID.
f_name_max	The maximum length of an object name.

Fields for which a **vfs** structure has no values are set to 0 (zero).

Execution Environment

The **vfs_statfs** subroutine can be called from the process environment only.

Return Values

0	Success
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Files

/usr/include/sys/vfs.h	Defines the **vfs** structure.
/usr/include/sys/statfs.h	Defines the **statfs** structure.
/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **statfs** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Data Structures and Header Files for Virtual File Systems, Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vfs_sync Subroutine

Purpose

Forces a virtual file system (VFS) update to permanent storage.

Syntax

int vfs_sync ()

Description

The **vfs_sync** subroutine forces all data associated with a particular virtual file system type to be written to its storage. This subroutine is used to establish a known consistent state of the data.

Note: Unlike all the other VFS-related operations, the **vfs_sync** subroutine does not apply to a particular virtual file system, but rather to the generic file system (GFS) type.

Execution Environment

The **vfs_sync** subroutine can be called from the process environment only.

Return Values

0	Success
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **sync** subroutine.

File System Overview, List of Virtual File System Operations, Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vfs_umount Subroutine

Purpose

Unmounts a virtual file system.

Syntax

int vfs_umount (*vfsp*)
**struct vfs ****vfsp*;

Parameter

vfsp	Points to the **vfs** structure being unmounted. This structure is defined in the **<sys/vfs.h>** file.

Description

The **vfs_umount** subroutine unmounts 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 do the following:

- Guarantee the syntax of the **vumount** 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 (vnode), as well as write permission to the file represented by the mounted-over vnode.
- Ensure that the virtual file system being unmounted contains no mount points for other virtual file systems.
- Ensure that the root vnode is not in use except for the mount. The root vnode is also referred to as the mounted vnode.
- Clear the **v_mvfsp** field in the stub vnode. 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** subroutine calls free the root vnode. An error return from the **vfs_umount** subroutine causes the mount point to be re_established. A 0 (zero) returned from the **vfs_umount** subroutine indicates the routine was successful and that the **vfs** structure was released.

Execution Environment

The **vfs_umount** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Files

/usr/include/sys/vmount.h	Describes types of file systems for the **vmount** subroutine.
/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **umount** subroutine.

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding Data Structures and Header Files for Virtual File Systems, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts*.

vfs_vget Subroutine

Purpose

Converts a file identifier into a virtual node (vnode).

Syntax

int vfs_vget (*vfsp, vpp, fidp***)**
struct vfs **vfsp*;
struct vnode ***vpp*;
struct fileid **fidp*;

Parameters

vfsp	Points to the virtual file system that is to contain the vnode. Any returned vnode should belong to this virtual file system.
vpp	Points to the place to return the vnode pointer. This is set to point to the new vnode. The fields in this vnode should be set as follows:

v_vntype	The type of vnode 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.

If the *fidp* parameter is invalid, the *vpp* parameter should be set to NULL by the **vfs_vget** subroutine.

fidp	Points to a file identifier. This is a file system-specific file identifier that must conform to the **fileid** structure.

Description

The **vfs_vget** subroutine converts a file identifier into a vnode. This subroutine uses information in the *vfsp* and *fidp* parameters to create a vnode or attach to an existing vnode. This vnode represents, logically, the same file system object as the file identified by the *fidp* parameter.

If the vnode already exists, successful operation of this subroutine increments the vnode use count and returns a pointer to the vnode. If the vnode does not exist, the **vfs_vget** subroutine creates it using the **vn_get** kernel service and returns a pointer to the new vnode.

Execution Environment

The **vfs_vget** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.
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.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), The Virtual File System Kernel Services, Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts*.

vn_access Subroutine

Purpose

Validates user access to a virtual node (vnode).

Syntax

int vn_access (*vp*, *mode*, *who***)**
**struct vnode ****vp*;
int *mode*;
int *who*;

Parameters

vp	Points to the vnode.
mode	Identifies the access mode.
who	Specifies what ID to check the access against. This parameter should be one of the following values which are defined in the **<sys/access.h>** file:

ACC_SELF	Determines if access is permitted for the current process. The effective user and group IDs, and the concurrent group set 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.

Description

The **vn_access** subroutine validates user access to a vnode. This subroutine is used to implement the **access** subroutine. The vnode is held for the duration of the **vn_access** subroutine. The vnode count is unchanged by this subroutine.

In addition, the **vn_access** subroutine is used for permissions checks from within the file system implementation. The valid types of access are listed in the **<sys/access.h>** file. Current modes are read, write, execute, and existence check.

Note: The **vn_access** subroutine must ensure that write access is not requested on a read-only file system.

Execution Environment

The **vn_access** subroutine can be called from the process environment only.

Return Values

0	Success.
EACCESS	No access is allowed.
ERRNO	Error number from **<sys/errno.h>** file on failure.

Files

/usr/include/access.h	Lists the valid types of access.
/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **access** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_close Subroutine

Purpose

Releases the resources associated with an open virtual node (vnode).

Syntax

int vn_close (*vp*, *flag*, *vinfo***)**
**struct vnode ****vp*;
int *flag*;
caddr_t *vinfo*;

Parameters

vp	Points to the vnode.
flag	Identifies the flag word from the file pointer.
vinfo	Specifies file system-specific information for the vnode.

Description

The **vn_close** subroutine releases the resources associated with an open vnode. Any file system–specific close operations are also performed. Normally, the actual release of resources is reserved till the last close of the vnode. The release of resources is handled in the **vn_rele** subroutine.

A **vn_close** subroutine is called only when the use count of an associated file structure entry goes to zero.

Note: The vnode is held over the duration of the **vn_close** subroutine.

Execution Environment

The **vn_close** subroutine can be called from the process environment only.

Return Values

Close errors are file system dependent.

Related Information

The **close** subroutine, **vn_open** subroutine, **vn_rele** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_create Subroutine

Purpose

Creates a new file.

Syntax

int vn_create (*dp*, *vpp*, *flag*, *pname*, *mode*, *vinfop*)
struct vnode **dp*;
struct vnode ***vpp*;
int *flag*;
char **pname*;
int *mode*;
caddr_t **vinfop*;

Parameters

dp	Points to the virtual node (vnode) of the parent directory.
vpp	Points to the place in which the pointer to a vnode for the newly created file is returned.
flag	Specifies an integer flag word. This parameter is used by the **vn_create** subroutine to open the file.
pname	Points to the name of the new file.
mode	Specifies the mode for the new file.
vinfop	Records information about the open procedure for the file system implementation. This information is supplied to subsequent vnode operations and is file system-specific.

Description

The **vn_create** subroutine creates a VREG type of vnode in the directory specified by the *dp* parameter. (Other vnode operations create directories and special files.) Virtual node types are defined in the **<sys/vnode.h>** file. The vnode of the parent directory is held during the processing of the **vn_create** subroutine.

To create a file, the **vn_create** subroutine 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** subroutine before calling the **vn_create** subroutine.

Execution Environment

The **vn_create** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines node operations and structures.

Related Information

The **vn_lookup** subroutine.

Special Files Overview in *Files Reference*.

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts*.

vn_fclear Subroutine

Purpose

Releases portions of a file.

Syntax

int vn_fclear (*vp*, *flags*, *offset*, *len*, *vinfo*)
struct vnode **vp*;
int *flags*;
off_t *offset*;
ulong *len*;
caddr_t *vinfo*;

Parameters

vp	Points to the virtual node (vnode) of the file.
flags	Identifies the flags from the open file structure.
offset	Indicates where to start clearing in the file. This parameter is updated to reflect the number of bytes cleared.
len	Specifies the length of the area to be cleared.
vinfo	Indicates data from the open file structure specific to the virtual file system.

Description

The **vn_fclear** subroutine clears bytes in a file, returning whole free blocks to the underlying file system. This subroutine performs the clear regardless of whether the file is mapped.

Execution Environment

The **vn_fclear** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **fclear** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_fid Subroutine

Purpose

Builds a file identifier for a virtual node (vnode).

Syntax

int vn_fid (*vp*, *fidp***)**
struct vnode **vp*;
struct fileid **fidp*;

Parameters

vp	Points to the vnode that requires the file identifier.
fidp	Points to where to return the file identifier.

Description

The **vn_fid** subroutine builds a file identifier for a vnode.

Execution Environment

The **vn_fid** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_fsync Subroutine

Purpose

Flushes information in memory and data to disk.

Syntax

int vn_fsync (*vp*, *flags***)**
**struct vnode ****vp*;
int *flags*;

Parameters

vp	Points to the virtual node (vnode) of the file.
flags	Identifies flags from the open file.

Description

The **vn_fsync** subroutine flushes information in memory and data about the file to permanent storage.

Execution Environment

The **vn_fsync** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure. Possible errors are specific to the virtual file system.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **fsync** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_ftrunc Subroutine

Purpose

Truncates a file.

Syntax

int vn_ftrunc (*vp*, *flags*, *length*, *vinfo***)**
**struct vnode ****vp*;
int *flags*;
ulong *length*;
caddr_t *vinfo*;

Parameters

vp	Points to the virtual node (vnode) of the file.
flags	Identifies flags from the open file structure.
length	Specifies the length to which the file should be truncated.
vinfo	Contains data from the open file structure specific to the virtual file system. This data is used as specified by the file system.

Description

The **vn_ftrunc** subroutine decreases the length of a file by truncating it. This operation fails if any process other than the caller has locked a portion of the file past the specified offset.

Execution Environment

The **vn_ftrunc** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **ftruncate** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_getacl Subroutine

Purpose

Retrieves the access control list (ACL) for a file.

Syntax

#include <sys/acl.h>

int vn_getacl (*vp*, *uiop***)**
**struct vnode ***vp*;
**struct uio ***uiop*;

Parameters

vp	Specifies the virtual node (vnode) of the file system object.
uiop	Specifies the **uio** structure that defines the storage for the ACL.

Description

The **vn_getacl** subroutine retrieves the ACL.

Execution Environment

The **vn_getacl** subroutine can be called from the process environment only.

Return Values

0	Indicates a successful operation.
ENOSPC	Indicates that the buffer size specified in the *uiop* parameter was not large enough to hold the ACL. If this is the case, then the first word of the user's buffer (data in the **uio** structure specified by the *uiop* parameter) is set to the appropriate size.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **accessx** subroutine, **chacl** subroutine, **chmod** subroutine, **chown** subroutine, **statacl** subroutine.

The **iaccess** kernel service, **iowner** kernel service.

File System Overview, List of Virtual File System Operations, Understanding the uio Structure, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_getattr Subroutine

Purpose

Gets the attributes of a file.

Syntax

int vn_getattr (*vp*, *vap***)**
struct vnode ****vp*;**
struct vattr ****vap*;**

Parameters

vp	Points to the virtual node (vnode) for the file.
vap	Points to a **vattr** structure.

Description

The **vn_getattr** subroutine fills in the **vattr** structure. If this subroutine succeeds, the **vattr** structure indicated by the *vap* parameter contains all the relevant attributes of the file. The **vattr** structure is defined in the **<sys/vattr.h>** file.

Note: The indicated vnode is held for the duration of the **vn_getattr** subroutine.

Execution Environment

The **vn_getattr** subroutine can be called from the process environment only.

Return Values

0	Success
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Files

/usr/include/sys/vattr.h	Describes the attributes of files across all types of supported file systems.
/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **statx** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_hold Subroutine

Purpose

Assures that a virtual node (vnode) is not destroyed.

Syntax

int vn_hold (*vp***)**
**struct vnode ****vp*;

Parameter

vp	Points to the vnode.

Description

The **vn_hold** subroutine increments the **v_count** field, the hold count on the vnode, and the vnode's underlying gnode (generic node). This incrementation assures that the vnode is not deallocated.

Execution Environment

The **vn_hold** subroutine can be called from the process environment only.

Return Values

0	Success
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the vnode structure and vnode operations.

Related Information

File System Overview, List of Virtual File System Operations, Understanding Generic Inodes (Gnodes), Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts*.

vn_ioctl Subroutine

Purpose

Performs miscellaneous operations on special files.

Syntax

int vn_ioctl (*vp, cmd, arg, flags, ext***)**
**struct vnode ****vp*;
int *cmd*;
caddr_t *arg*;
int *flags*;
int *ext*;

Parameters

vp	Points to the virtual node (vnode) on which to perform the operation.
flags	Identifies flags from the open file structure.
cmd	Identifies the specific command. Common operations for the **ioctl** subroutine are defined in the **<sys/ioctl.h>** header 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).
ext	Specifies the extended parameter passed by the **ioctl** subroutine. The **ioctl** subroutine always sets the *ext* parameter to 0 (zero).

Description

The **vn_ioctl** subroutine performs miscellaneous operations on special files. These files are usually devices, but no restrictions (besides those specific to virtual file systems) are placed on the type of files the **vn_ioctl** subroutine can be applied to.

Execution Environment

The **vn_ioctl** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.
EINVAL	The file system does not support the subroutine.

Files

/usr/include/sys/ioctl.h	Contains **ioctl** definitions.
/usr/include/sys/vnode.h	Defines the vnode structure and vnode operations.

Related Information

The **ioctl** subroutine.

Special Files Overview in *Files Reference.*

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_link Subroutine

Purpose

Creates a hard link to a file.

Syntax

int vn_link (*vp*, *dp*, *name*)
struct vnode **vp*;
struct vnode **dp*;
caddr_t **name*;

Parameters

vp	Points to the virtual node (vnode) to link to. This vnode is held for the duration of the linking process.
dp	Points to the vnode for the directory in which the link is created. This vnode is held for the duration of the linking process.
name	Identifies the new name of the entry.

Description

The **vn_link** subroutine creates a hard link to a file. The logical file system ensures that the *dp* and *vp* parameters reside in the same virtual file system and that it is not a read-only file system.

Execution Environment

The **vn_link** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts*.

vn_lockctl Subroutine

Purpose

Sets, checks, and queries locks.

Syntax

int vn_lockctl (*vp, offset, lckdat, cmd, retry_fn, retry_id***)**
struct vnode ****vp*;**
off_t *offfset***;**
struct flock ****lckdat*;**
int *cmd***;**
int (**retry_fn***)();**
caddr_t *retry_id***;**

Parameters

vp	Points to the file's virtual node (vnode).
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 **vlock** structure. This structure describes the lock operation to perform.
cmd	Identifies the type of lock operation the **vn_lockctl** subroutine is to perform. It is a bit mask that takes the following lock-control values:

SETFLCK	If set, perform a lock set or clear. If clear, return the lock information. The **l_type** field in the **flock** 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 NULL, the **vn_lockctl** routine 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.)

Description

The **vn_lockctl** subroutine implements record locking. This subroutine uses the information in the **flock** structure to implement record locking.

If a requested lock is blocked by an existing lock, the **vn_lockctl** subroutine should establish a sleeping lock with the retry subroutine address (specified by the *retry_fn* parameter) stored in the subroutine. The **vn_lockctl** subroutine 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.

flock Structure

The **flock** structure is defined in the **<sys/flock.h>** file and includes the following fields:

l_type	Type of lock. This field takes the following values:

F_RDLCK	Read lock.
F_WRLCK	Write lock.
F_UNLCK	Unlock this record. A value of F_UNLCK starting at 0 (zero) till 0 for a length of 0 means unlock all locks on this file. Unlocking is done automatically when a file is closed.

l_whence	Location that the **l_start** field offsets.
l_start	Offset from the **l_whence** field.
l_len	Length of record. If this field is 0 then the remainder of the file is specified.
l_vfs	Virtual file system that contains the file.
l_sysid	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** subroutine.
l_pid	Process ID (PID) of the lock owner. This field must be filled before the call to the **vn_lockctl** subroutine.

Execution Environment

The **vn_lockctl** subroutine can be called from the process environment only.

Return Values

0	Success.
EAGAIN	A blocking lock exists and the caller did not use the **SLPFLCK** flag to request that the operation sleep .
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Files

/usr/include/sys/flock.h	Identifies the data type of the file segment-locking set.
/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_lookup Subroutine

Purpose

Finds an object by name in a directory.

Syntax

int vn_lookup (*dvp, vpp, name, flags***)**
**struct vnode ****dvp*;
**struct vnode *****vpp*;
**char ****name*
int *flags*;

Parameters

dvp	Points to the virtual node (vnode) of the directory to be searched. The logical file system verifies that this vnode is of type VDIR.
flags	Specifies lookup directives, including these three flags:

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 place to which to return the vnode pointer, if the pointer is found. Otherwise, NULL should be placed in this memory location.
name	Points to a null-terminated character string containing the file name to look up.

Description

The **vn_lookup** vnode operation is used to find a vnode, given a name in a known directory. This routine returns the vnode specified into the indicated directory.

The use count in the *dvp* vnode 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 vnode is placed in the memory location specified by the *vpp* parameter, and the vnode's hold count is incremented. (In this case, this routine returns 0.) If the file name is not found, NULL is placed in the *vpp* parameter, and the function returns ENOENT. Errors are reported with a return code from the **<sys/errno.h>** file. Possible errors are usually specific to the particular virtual file system involved.

Execution Environment

The **vn_lookup** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **sys/errno.h** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_map Subroutine

Purpose

Performs file system-specific operations when a file is mapped.

Syntax

int vn_map (*vp*, *addr*, *length*, *offset*, *flags*)
struct vnode **vp*;
caddr_t *addr*;
uint *offset*;
uint *length*;
uint *flags*;

Parameters

vp	Points to the virtual node (vnode) of the file.
addr	Identifies the location within the process's 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 **<sys/shm.h>** file. The following masks 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 mask is set, updates to the segment are deferred until a 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** subroutine is also responsible for calling the **vn_fsync** subroutine.

Note: Mapped segments do not reflect modifications made to a copy-on-write segment.

Description

The **vn_map** subroutine performs file system-specific operations when a file is mapped. 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** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Files

/usr/include/sys/shm.h	Defines IPC-shared memory segments.
/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **shmat** subroutine, **vn_fsync** subroutine.

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_mkdir Subroutine

Purpose

Creates a directory.

Syntax

int vn_mkdir (*dp*, *name*, *mode*)
struct vnode **dp*;
caddr_t *name*;
int *mode*;

Parameters

dp	Points to the virtual node (vnode) of the new directory's parent directory. This vnode is held for the duration of the subroutine.
name	Specifies the name of the new directory.
mode	Specifies the permission modes of the new directory.

Description

The **vn_mkdir** subroutine creates a new directory. The logical file system ensures that the *dp* parameter does not reside on a read-only file system.

Execution Environment

The **vn_mkdir** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **mkdir** subroutine.

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_mknod Subroutine

Purpose

Creates any type of file.

Syntax

int vn_mknod (*dvp*, *name*, *mode*, *dev***)**
**struct vnode ****dvp*;
**caddr_t ****name*;
int *mode*;
dev_t *dev*;

Parameters

dvp	Points to the virtual node (vnode) for the directory to contain the new file. This vnode is held for the duration of the **vn_mknod** subroutine.
name	Specifies the name of the new file.
mode	Identifies the integer mode that indicates the type of file and its permissions.
dev	Identifies an integer device number.

Description

The **vn_mknod** subroutine creates any type of file. The file is created with the specified *mode* parameter. If this is a special file, the device number is as specified by the *dev* parameter.

The logical file system verifies that the *dvp* parameter does not reside in a read-only file system.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Related Information

The **mknod** subroutine.

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_open Subroutine

Purpose

Opens a file for reading or writing.

Syntax

int vn_open (*vp*, *flag*, *ext*, *vinfop***)**
struct vnode **vp*;
int *flag*;
caddr_t *ext*;
caddr_t *vinfop*;

Parameters

vp	Points to the virtual node (vnode) associated with the desired file. The vnode is held for the duration of the open process.
flag	Specifies the type of access. Access modes are defined in the **<fcntl.h>** file. **Note:** The **vn_open** subroutine does not use the FCREAT mode.
ext	Points to external data. This parameter is used if the subroutine is opening a device.
vinfop	May be used to record information about the open process. The information recorded by this parameter is supplied to subsequent vnode operations.

Description

The **vn_open** subroutine initiates a process's access to a vnode (virtual node) and its underlying file system object. The operation of the **vn_open** subroutine varies between virtual file system (VFS) implementations. A successful **vn_open** subroutine must leave a vnode count of at least 1 (one).

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** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Files

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.
/usr/include/sys/fcntl.h	Defines access modes.

Related Information

The **open** subroutine, **vn_close** subroutine.

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_rdwr Subroutine

Purpose

Performs file I/O.

Syntax

int vn_rdwr (*vp*, *op*, *flags*, *uiop*, *ext*, *vinfo***)**
**struct vnode ****vp*;
enum uio_rw *op*;
int *flags*;
**struct uio ****uiop*;
int *ext*;
caddr_t *vinfo*;

Parameters

vp	Points to the file's virtual node (vnode).
op	Specifies an enum that indicates a read or write operation. This parameter takes the following values: • UIO_READ • UIO_WRITE. These values are found in the **<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	Contains file system-specific data from the open file structure. This information is stored by a **vn_open** or **vn_create** subroutine. The use of this data is determined by the file system implementation.

Description

The **vn_rdwr** subroutine reads or writes data from or to an object represented by a vnode. The **vn_rdwr** subroutine 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** subroutine can be called from the process environment only.

Files

/usr/include/sys/uio.h	Describes the desired data transfer.
/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Related Information

The **read** subroutine, **vn_create** subroutine, **vn_open** subroutine, **write** subroutine.

File System Overview, List of Virtual File System Operations, Understanding the uio StructureUnderstanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts*.

vn_readdir Subroutine

Purpose

Reads directory entries in standard format.

Syntax

int vn_readdir (*vp***,** *uiop***)**
struct vnode ****vp*;**
struct uio ****uiop*;**

Parameters

vp	Points to the directory's virtual node (vnode).
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.

Description

The **vn_readdir** subroutine is used to access directory entries in a standard way. These directories should be returned as an array of **dirent** structures. The **<sys/dir.h>** file contains the definition of a **dirent** structure.

The **vn_readdir** subroutine 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.

End-of-file 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** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure. Possible errors are specific to the virtual file system.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **readdir** subroutine.

File System Overview, List of Virtual File System Operations, Understanding the uio Structure, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_readlink Subroutine

Purpose

Reads the contents of a symbolic link.

Syntax

int vn_readlink (*vp, uio***)**
struct vnode **vp*;
struct uio **uio*;

Parameters

vp	Points to a virtual node (vnode) structure. The **vn_readlink** subroutine holds this vnode 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** subroutine.

Description

The **vn_readlink** subroutine reads and returns the contents of a symbolic link. The logical file system finds the vnode for the symbolic link, so this routine simply reads the data blocks for the symbol link.

Execution Environment

The **vn_readlink** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding the uio Structure, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts*.

vn_rele Subroutine

Purpose

Releases a virtual node (vnode).

Syntax

int vn_rele (*vp***)**
**struct vnode ***vp*;

Parameter

vp	Points to the vnode.

Description

The **vn_rele** subroutine releases the object associated with a vnode. If the object was the last reference to the vnode, the **vn_rele** subroutine then calls the **vn_free** kernel service to deallocate the vnode.

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 vnode to be released is the last vnode on the chain of the **vfs** structure, then the virtual file system must be deallocated with the **vn_rele** subroutine.

Execution Environment

The **vn_rele** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Related Information

The **vn_free** kernel service.

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding the uio Structure, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_remove Subroutine

Purpose

Removes a file or directory.

Syntax

int vn_remove (*vp*, *dvp*, *name***)**
struct vnode ****vp*;**
struct vnode ****dvp*;**
char ****name*;**

Parameters

vp	Points to a virtual node (vnode). The vnode indicates which file to remove and is held over the duration of the **vn_remove** subroutine.
dvp	Points to the vnode of the parent directory. This directory contains the file to be removed. The directory's vnode is held for the duration of the **vn_remove** subroutine.
name	Identifies the name of the file.

Description

The **vn_remove** subroutine removes an entry (or link) for a file from a directory.

The logical file system assumes that the **vn_remove** subroutine calls the **vn_rele** subroutine. 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** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **unlink** subroutine, **vn_rele** subroutine.

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding the uio Structure, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_rename Subroutine

Purpose

Renames a file or directory.

Syntax

int vn_rename (*srcvp*, *srcdvp*, *oldname*, *destvp*, *destdvp*, *newname***)**
**struct vnode ****srcvp*;
**struct vnode ****srdcvp*;
**char ****oldname*;
**struct vnode ****destvp*;
**struct vnode ****desdvp*;
**char ****newname*;

Parameters

srcvp	Points to the virtual node (vnode) of the object to rename.
srcdvp	Points to the vnode 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 vnode of the new object. This pointer is used only if the new object exists. Otherwise, this parameter is NULL.
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.

Description

The **vn_rename** subroutine renames a file or directory. This routine provides for the following 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 doesn't yet exist in a different parent directory.
- Renames an old object to a new object that exists in the same parent directory.
- Renames an old objec to a new object that doesn't yet exist in the same parent directory.

To ensure that this routine executes correctly, the logical file system guarantees the following:

- File names are not renamed across file systems.
- The old and new object (if specified) are not the same.
- The old and new parent directories are of the same type of vnode.

Execution Environment

The **vn_rename** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **rename** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Logical File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_revoke Subroutine

Purpose

Revokes all access to an object.

Syntax

int vn_revoke (*vp*, *cmd*, *flag*, *vinfop*)
struct vnode **vp*;
int *cmd*;
int *flag*;
caddr_t *vinfop*;

Parameters

vp	Points to the virtual node (vnode) 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 (one).

flag	Identifies the flags from the **file** structure.
vinfop	Contains file system-specific information from the **file** structure.

Description

The **vn_revoke** subroutine revokes further access to an object.

Execution Environment

The **vn_revoke** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **frevoke** subroutine, **revoke** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_rmdir Subroutine

Purpose

Removes a directory.

Syntax

int vn_rmdir (*vp, dp, name***)**
**struct vnode ****vp*;
**struct vnode ****dp*;
**char ****pname*;

Parameters

vp	Points to the vnode of the directory.
dp	Points to the parent of the directory to remove.
pname	Points to the name of the directory to remove.

Description

The **vn_rmdir** subroutine removes 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' vnodes (virtual nodes) are held for the duration of the routine.

Execution Environment

The **vn_rmdir** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **rmdir** subroutine.

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_select Subroutine

Purpose

Polls a virtual node (vnode) for immediate I/O.

Syntax

int vn_select (*vp*, *correl*, *e*, *re*, *notify*, *vinfo***)**
struct vnode ****vp*;**
int *correl***;**
int *e***;**
int *re***;**
int (**notify***)();**
caddr_t *vinfo***;**

Parameters

vp	Points to the vnode to be polled.
correl	Specifies the ID used for correlation in the **selnotify** kernel service.
re	Returns an events list. If the vnode is ready for immediate I/O, this field should be set to indicate the requested event is ready.
e	Identifies the requested event.
notify	Specifies the subroutine to call when the event occurs. This parameter is for nested polls.
vinfo	Identifies file system-specific information from the file structure.

Description

The **vn_select** subroutine polls a vnode to determine if it is immediately ready for I/O. This subroutine implements the **select** and **poll** subroutines.

File system implementations can support constructs, such as devices or pipes that support the select sematics. The **fp_select** kernel service provides more information about select and poll requests.

Execution Environment

The **vn_select** subroutine can be called from the process environment only.

Return Values

0	Success
ERRNO	Error number from the **<sys/errno.h>** on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **poll** subroutine, **select** subroutine.

The **fp_select** kernel service, **selnotify** kernel service.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_setacl Subroutine

Purpose

Sets the access control list for a file.

Syntax

#include <sys/acl.h>

int vn_setacl (*vp*, *uiop***)**
**struct vnode ****vp*;
**struct uio ****uiop*;

Parameters

vp	Specifies the virtual node (vnode) of the file system object.
uiop	Specifies the **uio** structure that defines the storage for the call arguments.

Description

The **vn_setacl** subroutine sets the access control list.

Execution Environment

The **vn_setacl** subroutine can be called from the process environment only.

Return Values

0	Indicates a successful operation.
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.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

The **accessx** subroutine, **chacl** subroutine, **chmod** subroutine, **chown** subroutine, **statacl** subroutine.

The **iaccess** kernel service, **iowner** kernel service.

File System Overview, List of Virtual File System Operations, Understanding the uio Structure, Understanding Virtual Nodes (Vnodes) in *Kernel Extensions and Device Support Programming Concepts.*

vn_setattr Subroutine

Purpose

Sets attributes of a file.

Syntax

```
int vn_setattr (vp, cmd, arg1, arg2, arg3)
struct vnode *vp;
   int cmd;
   int arg1;
   int arg2;
   int arg3;
```

Parameters

vp	Points to the vnode (virtual node) of the file.
cmd	Defines the setting operation. This parameter takes the following values:

V_OWN	Sets the UID and GID to the UID and GID values of the new file owner.
V_UTIME	Sets the access and modification time for the new file.
V_MODE	Sets the file mode.

The **<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 the **vn_setattr** subroutine is called.

Description

The **vn_setattr** subroutine sets the attributes of a file. This subroutine is used to implement the chmodx, chownx, and utime subroutines.

The values that the *arg* parameters take depend on the *cmd* parameter with which the **vn_setattr** subroutine is called. The **vn_setattr** subroutine accepts the following *cmd* values and *arg* parameters:

Possible cmd Values for the vn_setattr Routine

Command	V_OWN	V_UTIME	V_MODE
arg1	**int** *flag*;	**int** *flag*;	**int** *mode*;
arg2	**int** *uid*;	**time_t** *atime*;	Unused
arg3	**int** *gid*;	**time_t** *mtime*;	Unused

V_OWN	Sets the UID and GID to the 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. A *flag* value of T_SETTIME indicates that specific values have not been provided and the access and modification times of the object should beset to current system time. If T_SETTIME is not specified, the values are specified by the *atime* and *mtime* variables.
V_MODE	Sets the file mode.

Execution Environment

The **vn_setattr** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from **<sys/errno.h>** on failure.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and **vnode** operations.

Related Information

The **chmodx** subroutine, **chownx** subroutine, **utime** subroutine.

File System Overview, Understanding Virtual Nodes (Vnodes) in *Kernel Extensions and Device Support Programming Concepts.*

vn_strategy Subroutine

Purpose

Accesses blocks of a file.

Syntax

int vn_strategy (*vp*, *bp***)**
struct vnode ****vp*;**
struct buf ****bp*;**

Parameters

vp	Points to the virtual node (vnode) of the file.
bp	Points to a **buf** structure that describes the buffer.

Description

The **vn_strategy** subroutine accesses blocks of a file. This subroutine is intended to provide a block-oriented interface for servers for efficiency in paging.

Return Values

0	Success
ERRNO	Error number from the **<sys/errno.h>** file on failure. Possible errors are specific to the virtual file system.

Related Information

File System Overview, List of Virtual File System Operations, Understanding the uio Structure, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_symlink Subroutine

Purpose

Creates a symbolic link.

Syntax

int vn_symlink (*vp, linkname, target***)**
**struct vnode ****vp*;
**char ****linkname*;
**char ****target*;

Parameters

dp	Points to the virtual node (vnode) 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.

Description

The **vn_symlink** subroutine creates 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** subroutine can be called from the process environment only.

Return Values

0	Success.irtual
ERRNO	Error number from the **<sys/errno.h>** file on failure.

Related Information

The **symlink** subroutine.

File System Overview, List of Virtual File System Operations, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

vn_unmap Subroutine

Purpose

Unmaps a file.

Syntax

int vn_unmap (*vp*, *flag***)**
struct vnode **vp*;
ulong *flag*;

Parameters

vp	Points to the file's virtual node (vnode).
flag	Indicates how the file was mapped. This flag takes the following values: • SHM_RDONLY • SHM_COPY.

Description

The **vn_unmap** subroutine unmaps a file. When this 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 do only those operations that are unique to the file system. The logical file system handles virtual-memory management operations.

Execution Environment

The **vn_unmap** subroutine can be called from the process environment only.

Return Values

0	Success.
ERRNO	Error number from the **<sys/errno.h>** file on failure. Possible errors are specific to the virtual file system implementation.

File

/usr/include/sys/vnode.h	Defines the **vnode** structure and vnode operations.

Related Information

File System Overview, List of Virtual File System Operations, Logical File System Overview, Understanding Virtual Nodes (Vnodes), Virtual File System Overview in *Kernel Extensions and Device Support Programming Concepts.*

Part 2. Extending Device Subsystems

Chapter 4. Configuration Subsystem

Machine Device Driver

Description

The machine device driver provides an interface to platform-specific hardware for the AIX configuration and RAS subsystems. The machine device driver supports two special files for accessing this hardware from user mode: **/dev/nvram/***n* and **/dev/bus0.** The **/dev/nvram** special file provides access to special nonvolatile RAM or ROS for the purposes of storing or retrieving error information and system boot information. The **/dev/bus0** special file provides access to the I/O bus for system configuration and diagnostic purposes. The presence and use of this device driver and its associated special files are platform-specific and should not be used by general applications.

Programs attempting to open the **/dev/nvram** or **/dev/bus0** special file must have the appropriate privilege.

Driver Initialization and Termination

There are no special initialization and termination requirements for the machine device driver. This driver is statically bound with the AIX Operating System kernel and is initialized during kernel initialization. This device driver does not support termination and may not be unloaded.

/dev/nvram Special File Support

The open and close Subroutines

The machine device driver supports the **/dev/nvram** special file as a multiplexed character special file. This special file and the support for NVRAM is provided only on the RISC System/6000 hardware platform to support the AIX configuration and RAS subsystems. These subsystems open the **/dev/nvram/***n* special file with a channel name *n* specifying the data area to be accessed. An exception is channel 0, which does not access a data area. Instead, it provides access to general NVRAM control functions and the LED display on the RISC System/6000 front panel. Channels 1 to 99 are supported for system boot, configuration, and RAS subsystems.

The following two channels are handled as a special case by the machine device driver during a **close** operation:

n=1	Custom Boot Device Driver #1 code area
n=2	Custom Boot Device Driver #2 code area .

These two channels are used to support the extendable boot device function provided by the RISC System/6000 boot ROS. This function allows a custom boot driver to be placed in NVRAM in order to boot from a device not originally supported by the system ROS. A custom boot device header must also be created in NVRAM so that the ROS can identify and validity-check the custom boot driver before loading and executing the code. This header must define the location, length, and CRC (cyclic redundancy code) value for the boot driver.

The machine device driver automatically generates this header whenever a **close** subroutine call has been issued to either channel 1 or 2. The machine device driver also verifies that the NVRAM data area for channel 1 and 2 is contiguous storage since the custom boot driver code is required (by the ROS) to be in contiguous NVRAM storage. If this is not the case, the **close** operation returns an error.

A special channel name of **pcbios** can be used to read the compressed PCAT BIOS code stored in nonvolatile memory on the RISC System/6000 platform. This compressed PCAT BIOS code may be uncompressed and used by an application program to assist in providing PC simulation on the RISC System/6000 platform.

A special channel name of **base** can be used to read the base customize information stored as part of the boot record. This information was originally copied to the disk by the **savebase** command and is only copied by the driver at boot time. Therefore, the base customize information does not change after subsequent invocations to the **savebase** routine.

Multiple concurrent opens to the same channel are not supported by the machine device driver and will return with an error.

The read and write Subroutines

The **read** and **write** subroutines are supported after a successful open of the **/dev/nvram/***n* special file for channel number *n* greater than 0. The **read** subroutine is also supported after a successful open of the **pcbios** and **base** channel. The **read** and **write** operations transfer data to and from the data area in NVRAM associated with the specified channel. The transfer starts at the offset (within the channel's data area) specified by the **offset** field associated with the file pointer used on the subroutine call.

On a read, if the end of the data area is reached before the transfer count is reached, the number of bytes read before the end of the data area was reached is returned. If the read starts at the end of the data area, zero bytes are read. If the read starts after the end of the data area, an error of ENXIO is returned by the driver.

For writes past the current end of data, additional NVRAM is allocated as necessary. If additional NVRAM cannot be allocated, an error of ENXIO is placed in the **errno** global variable and the number of bytes written before this condition occurred is returned. The **lseek** subroutine can be used to change the starting NVRAM data area offset to be used on a subsequent **read** or **write** call.

The ioctl Subroutine

The following **ioctl** operations can be issued to the machine device driver after a successful open of any channel using the **/dev/nvram** special file:

MIOIPLCB Returns the contents of the boot control block. The *arg* parameter is set to point to a **mach_dd_io** structure, which describes the data area where the boot control block has been placed. The format of this control block is specified in the **<sys/ioctl.h>** header file and the **mach_dd_io** structure is defined in the **<sys/mdio.h>** header file. This **ioctl** operation uses the following fields in the **mach_dd_io** structure:

- The **md_data** field points to a buffer at least the size of the value in the **md_size** field.
- The **md_size** field specifies the size (in bytes) of the buffer pointed to by the **md_data** field.

MIONVLED Writes the value found in the *arg* parameter to the RISC System/6000 system front panel LED display. On this panel, three digits are available and the *arg* parameter value can range from 0 to hex FFF. An explanation of the LED codes can be found in the **<sys/mdio.h>** header file.

MIONVALLOC Allocates the specified amount of NVRAM storage to the specified channel. The channel number and amount of NVRAM to be allocated are defined in the **mach_dd_io** structure pointed to by the *arg* parameter. This structure is defined in the **<sys/mdio.h>** header file. The following fields are used by this command:

- The **md_type** field specifies the channel number for which the storage is to be allocated.
- The **md_size** field specifies the number of bytes of NVRAM storage to be added to the data area for the specified channel.

The MIONVALLOC operation can be used to add storage to an existing data area. However, that data area will not typically be contiguous in NVRAM storage. This has no effect on data accessed through the read and write operations, but does affect data areas defined by channels 1 and 2. These channels define data areas accessed by system ROS and must be in contiguous NVRAM. The storage allocated by this operation is guaranteed to be contiguous in NVRAM.

This operation can be used in conjunction with the MIONVFREE operation to ensure that the data areas defined by channel 1 or 2 are in contiguous NVRAM.

MIONVFREE

Frees all NVRAM storage associated with the channel specified by the *arg* parameter. If the value of the *arg* parameter is 0, the NVRAM storage associated with the channel specified on the **open** subroutine is freed. This operation is typically used to create or update custom boot driver code in channel 1 or 2. This might be done to free an old area before allocating a new contiguous data area with the MIONVALLOC **ioctl** operation.

MIONVGET Reads data from a NVRAM address and returns data in the buffer provided by the caller. This is useful for reading the ROS area of NVRAM. A structure defining this area is in the **<sys/mdio.h>** header file.

MIONVPUT Writes data to a NVRAM address from the buffer provided by the caller. This operation is used only to update the ROS area of NVRAM and only by system commands. Use of this operation in other areas of NVRAM can cause unpredictable results to occur. If the NVRAM address provided is within the ROS area, a new cyclic redundancy code (CRC) for the ROS area is generated.

IOCINFO Returns machine device driver information in the caller's **devinfo** structure (pointed at by the *arg* parameter). This structure is defined in the **<sys/devinfo.h>** header file. The device type for this device driver is DD_PSEU.

Error Conditions

The following error conditions may be returned when accessing the machine device driver via the **/dev/nvram** special file:

ENOENT	An open for read access was requested for a channel that has not been allocated. This error code is also possible when a read was attempted to channel 0.
EBUSY	An open was requested for a channel already open.
EIO	A close was requested for channel 1 or 2 with a non-contiguous data area or the machine device driver was unable to build the custom boot device header required by system ROS.
EFAULT	A buffer specified by the caller was invalid on a **read**, **write**, or **ioctl** subroutine call.
EINVAL	Either a MIONVFREE **ioctl** operation was issued specifying an invalid channel or a write was attempted to channel 0.
ENXIO	A read was attempted past the end of the data area specified by the channel or a write was unable to complete due to insufficient NVRAM storage.
ENODEV	A write was attempted to the read-only **pcbios** channel.
ENOMEM	A request was made with a user-supplied buffer that is too small for the requested data.

/dev/bus0 Special File Support

The open and close Subroutines

The machine device driver supports the **/dev/bus0** special file as a character special file. This special file and support for access to the I/O bus and controller are provided on the RISC System/6000 hardware platform only to support the AIX configuration and diagnostic subsystems. The configuration subsystem accesses the I/O bus and controller through the machine device driver to determine the I/O configuration of the system. This driver can also be used to configure the I/O controller and devices as required for proper system operation. If the system diagnostics are unable to access a device through the diagnostic functions provided by the device's device driver, they may use the machine device driver to attempt further failure isolation.

The read and write Subroutines

The **read** and **write** subroutines are not supported by the machine device driver through the **/dev/bus0** special file and return an EINVAL return code in the **errno** global variable, if called.

The ioctl Subroutine

The /**dev/bus0** **ioctl** operations allow transfers of data between the system I/O controller or the system I/O bus and a caller-supplied data area. Because these **ioctl** operations use the **mach_dd_io** structure, the *arg* parameter on the **ioctl** subroutine must point to such a structure. The bus address, pointer to the caller's buffer and the length of the transfer is specified in the **mach_dd_io** structure. The **mach_dd_io** structure is defined in the **<sys/mdio.h>** header file and provides the following information:

- The **md_addr** field contains an I/O controller or I/O bus address.
- The **md_data** field points to a buffer at least the size of the value in the **md_size** field.
- The **md_size** field contains the number of bytes to be transferred.

The following commands can be issued to the machine device driver after a successful open of the /**dev/bus0** special file.

MIOBUSGET	Reads data from the I/O bus and returns it in a caller-provided buffer.
MIOBUSPUT	Writes data supplied in the caller's buffer to the I/O bus.
MIOCCGET	Reads data from the I/O controller and returns it in a caller-provided buffer.
MIOCCPUT	Writes data supplied in the caller's buffer to the I/O controller.
IOCINFO	Returns machine device driver information in the caller's **devinfo** structure, as specified by the *arg* parameter. This structure is defined in the **<sys/devinfo.h>** header file. The device type for this device driver is DD_PSEU.

Error Conditions

EFAULT	A buffer specified by the caller was invalid on an **ioctl** call.
EIO	An unrecoverable I/O error occurred on the requested data transfer.
ENOMEM	No memory could be allocated by the machine device driver for use in the data transfer.

Files

/dev/bus0

/dev/nvram/0, /dev/nvram/1, ... **/dev/nvram/***n*

Related Information

The **savebase** device configuration command.

The **bus** special file, **nvram** special file.

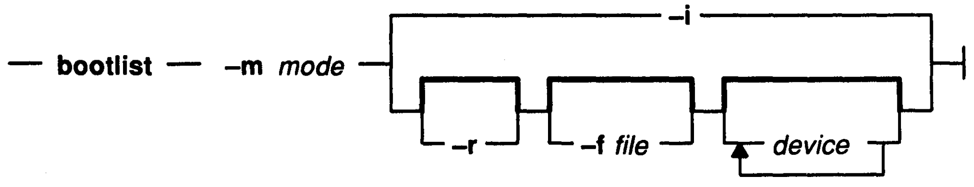
The **close** subroutine, **ioctl** subroutine, **lseek** subroutine, **open** subroutine, **read** subroutine, **write** subroutine.

bootlist Device Configuration Command

Purpose

Alters the list of boot devices (or the ordering of these devices in the list) available to the system.

Syntax



Description

The **bootlist** command allows the user to alter the list of boot devices available for system boot. This command can alter the contents of the battery-backed-up RAM boot device list and the choice of boot device used on the next and subsequent system boots. This command supports updating of the following:

- Service boot list. The service list designates possible boot devices when the front panel keylock switch is in the SERVICE position.
- Normal boot list. The normal list is used when the keylock is in the NORMAL position.
- Previous boot device entry. Retained in battery-backed-up RAM on the system unit.

The **bootlist** command supports the specification of generic device types as well as specific devices for boot candidates. Possible device names are listed either on the command line or in a file. Devices in the boot device list occur in the same order as devices listed on the invocation of this command.

It is strongly recommended that if more than one device is to be entered into the device list, the **-f** *file* flag be used. This makes an alterable record of the boot devices available for reference or future update. When the **-f** flag is used, the list of devices is taken from the file specified by the *file* variable. Devices from this list are then placed in the boot list in the order found in the file.

If the device list is not used, the system remembers the device previously used to boot the system (if it was not a diskette drive). If an additional boot device is added to the system, and no device list is used, the previous boot device may be invalidated by using this command. This forces the next re-boot to search for a boot device instead of using the previous boot device.

The selection of the boot device list to alter is made with the **-m** *mode* flag, where the *mode* variable is one of the keywords: **service, normal, both** or **prevboot**. If the **prevboot** keyword is specified, only the **-i** (invalidate) flag may be specified. If the **both** keyword is specified, then the service list and the normal list will contain exactly the same information. The **-i** flag invalidates the device list specified by the **-m** flag. The **-f** *file* flag can be used to specify device names in a file. Use of this option allows for future querying and updating of the devices listed in the file.

Device Choices

The device name specified on the command line (or in a file) can occur in one of two different forms:

- It can indicate a specific device by its device logical name.
- It can indicate a generic or special device type by keyword. The following generic device keywords are supported:

fd	Any standard I/O attached floppy diskette drives
scdisk	Any SCSI attached disk
badisk	Any direct bus attached disks (Model 320 only)
cd	Any SCSI attached CD-ROM
rmt	Any SCSI attached tape device.

The following special device keywords specify the use of a loadable boot device driver. This driver must have been previously loaded into the battery-backed-up RAM by the **nvload** command:

nvload1	NVRAM boot loader #1
nvload2	NVRAM boot loader #2.

When a specific device is to be included in the device list, the device's logical name (used with system management commands) must be specified. This logical name is made up of a prefix and a suffix. The suffix is generally a number and designates the specific device. The specified device must be in the AVAILABLE state. If it is not, the update to the device list is rejected and this command fails. The following devices and their associated logical names are supported (where the bold type is the prefix and the *xx* variable is the device-specific suffix):

fd*xx*	Floppy diskette device logical names
hdisk*xx*	Physical volume device logical names
cd*xx*	SCSI CD-ROM device logical names
rmt*xx*	Magnetic tape device logical names.

File Format When Using the -f Flag

The file specified by the *file* variable should contain device names separated by white space:

```
hdisk0 hdisk1 cd1
```

or one device per line:

```
hdisk0
hdisk1
cd1
```

Warning: Care must be taken in specifying the possible boot devices. A future reboot in NORMAL mode may fail if the devices specified in the device list become unbootable. A diskette boot is always available when the keylock is in the SERVICE position.

Warning: The system must not be powered off or reset during the operation of the bootlist command. If the system is reset, or power fails at a critical point in the execution of this command, a checksum error can cause the system setup information in battery-backed-up RAM to be lost.

Error Handling

If this command returns with an error, the device lists are not altered. The following device list errors are possible:

- If the user attempts to add too many devices to the boot list, the command fails indicating that too many devices were requested. The number of devices supported varies depending on the device selection.
- If an invalid keyword, invalid flag, or unknown device is specified, the command fails with the appropriate error message.
- If a specified device is not in the AVAILABLE state, the command fails with the appropriate error message.

Flags

device	Provides the names of the specific or generic devices to include in the boot list.
–f *file*	Indicates that the device information is to be read from the specified file name.
–i	Indicates that the device list specified by the **–m** option should be invalidated.
–m *mode*	Specifies which boot list to alter. Possible values for the *mode* variable are **normal**, **service**, **both**, or **prevboot**.
–r	Indicates that a hex dump of the specified device list in non-volatile RAM should be output after any specified alteration is performed. (This is normally used for problem determination.)

Examples

1. To invalidate the SERVICE mode boot list, enter:

   ```
   bootlist –m service –i
   ```

2. To make a boot list for NORMAL mode with devices listed on the command line, enter:

   ```
   bootlist –m normal hdisk0 hdisk1 rmt0 nvload1 fd
   ```

3. To make a boot list for NORMAL mode with a device list from a file, enter:

   ```
   bootlist  –m normal –f /bootlist.norm
   ```

   where /**bootlist.norm** is a file containing device names to be placed in the boot list for NORMAL mode. The device names in /**bootlist.norm** must comply with the described format.

4. To invalidate the previous boot device entry, enter:

   ```
   bootlist –m prevboot
   ```

Implementation Specifics

The **bootlist** command allows the user to alter the list of boot devices scanned by ROS (read-only storage) when the system is booted. Two device lists are stored in battery-backed-up RAM (NVRAM). One is for use when the front panel keylock is in the NORMAL position and the other when the keylock is in the SERVICE position. The **bootlist** command allows the user to update one or both of these lists.

Each list is a maximum of 84 bytes long. When searching for a boot device, the system ROS selects the first device in the list and determines if it is bootable. If no boot file system is detected on the first device, ROS moves on to the next device in the list. As a result, the ordering of devices in the device lists is extremely important.

If no device list has been supplied, or if it was empty, the ROS attempts to boot from the boot device used on a previous boot. (This assumes that the previous boot device was not a diskette drive.) If this boot device is unavailable or not bootable, the system ROS starts searching the I/O bus for the first device from which it can boot.

Related Information

The **nvload** command.

The **nvram** special file.

Device Configuration Commands.

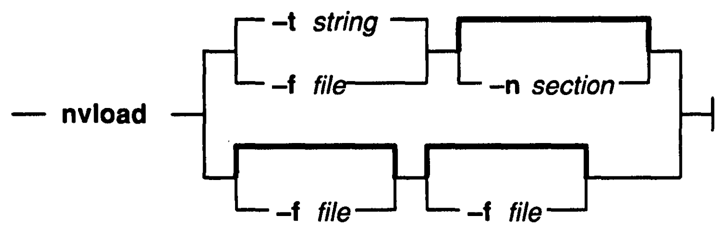
Machine Device Driver, SCSI Subsytem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts.*

nvload Device Configuration Command

Purpose

Loads a device driver into a section of NVRAM by reading an input device or the specified file or files.

Syntax



Description

When no parameters are specified, the **nvload** command uses the **tar** command to get the file to load. The **/dev/rfd0** device is the default device.

The specified file is loaded into NVRAM with the **–f** flag. If the **–n** flag is also used, then the specified file is loaded into the specified section of NVRAM.

If the **–f** flag occurs without the **–n** flag, then the NVRAM section used is dependent upon the order of the items on the command line in the following way. The first *file* variable specified with the **–f** flag is loaded into the first NVRAM section dedicated to device drivers. The second *file* variable is loaded into the second section.

This order also applies to the values in the **–t** flag when there is more than one file from the **tar** command. If there is only one file from tar, then the **–n** flag can be used.

Flags

–f *file*	Loads the specified file into NVRAM.
–n *section*	Specifies NVRAM section into which to load the specified file. Valid values for the *section* variable are 1 and 2.
–t *string*	Use the *string* variable as a string of **tar** command options. The string must include the **–xv** flags.

Examples

1. To load device drivers from the **/dev/rfd0** file using the **tar** command, enter:

   ```
   nvload
   ```

2. To load a specified driver from a specific device into a specific section of NVRAM, enter:

   ```
   nvload –t '–xvf/dev/rfd1/driver' –n 2
   ```

3. To load a specific file, enter:

   ```
   nvload –f driver
   ```

4. To load two drivers, enter:

   ```
   nvload –f /tmp/driver1 –f /u/guest/driver2
   ```

Related Information

The **tar** command.

The **nvram** special file.

Device Configuration Commands.

restbase Device Configuration Command

Purpose

Reads the base customized information from the boot image and restores it in the ODM.

Syntax

```
— restbase —|
```

Description

The **restbase** command reads the base customized information from the boot image. This command requires no operands and no output.

Related Information

The **savebase** command.

Device Configuration Commands.

Object Data Manager (ODM) Overview in *General Programming Concepts.*

Device Configuration Subsystem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts.*

savebase Device Configuration Command

Purpose

Saves information about base customized devices in the ODM onto the boot device.

Syntax

```
— savebase —|
```

Description

The **savebase** command backs up all of the information for base devices from the ODM onto the boot device. This command requires no operands and no output.

Related Information

The **restbase** command.

Device Configuration Commands.

Object Data Manager (ODM) Overview in *General Programming Concepts*.

attrval Device Configuration Subroutine

Purpose

Verifies that attribute values are within range.

Syntax

int attrval (*uniquetype*, *pattr*, *errattr*)
char **uniquetype*, **pattr*, ***errattr*;

Parameters

uniquetype	Identifies the predefined device object, which is a pointer to a character string of the form `class/subclass/type`.
pattr	Points to a character string containing the attribute-value pairs to be validated, in the form `attr1=val1 attr2=val2....`
errattr	Points to a pointer to a null-terminated character string. On return from the **attrval** subroutine, this string will contain the names of invalid attributes, if any are found. Each attribute name is separated from the next by spaces.

Description

The **attrval** subroutine is used to validate each of a list of input attribute values against the legal range. If no illegal values are found, this subroutine returns a value of 0. Otherwise, it returns the number of incorrect attributes.

If any of the attributes values are invalid, a pointer to a string containing a list of the names of the invalid attributes is returned in the *errattr* parameter. These attributes are separated by spaces.

Allocation of the error buffer is done in the **attrval** subroutine. However, a character pointer (for example, **char *errorb;**) must be declared in the calling routine. Thereafter, the address of that pointer is passed to the **attrval** subroutine (for example, `attrval(...,&errorb);`) as one of the parameters.

File

/lib/libcfg.a

Return Values

0	Indicates that all values are valid.
Nonzero	Indicates the number of erroneous attributes.

Related Information

Predefined Attribute Object Class, Customized Attribute Object Class, Predefined Devices Object Class.

Object Data Manager (ODM) Overview in *General Programming Concepts.*

Device Configuration Subroutines in *Kernel Extensions and Device Support Programming Concepts.*

genmajor Device Configuration Subroutine

Purpose

Generates the next available major number for a device driver instance.

Syntax

long genmajor (*device_driver_instance_name*)
char **device_driver_instance_name*;

Parameter

device_driver_instance_name	Points to a character string containing the device driver instance name.

Description

The **genmajor** device configuration subroutine is one of the routines designated for accessing the Customized Device Driver object class. If a major number already exists for the given device driver instance, then this major number is returned. Otherwise, a new major number is generated.

The **genmajor** subroutine creates an entry (object) in the Customized Device Driver object class for the major number information. The lowest available major number or the major number that has already been allocated is returned. The Customized Device Driver object class is locked exclusively by this routine until its completion.

File

/lib/libcfg.a

Return Values

If the **genmajor** subroutine executes successfully, a major number is returned. This major number is either the lowest available major number or the major number that has already been allocated to the device instance.

A value of –1 is returned if the **genmajor** subroutine fails.

Related Information

The **reldevno** device configuration subroutine, **relmajor** device configuration subroutine.

Customized Device Driver object class.

List of ODM Subroutines, Device Configuration Subroutines, Understanding Major and Minor Numbers in *Kernel Extensions and Device Support Programming Concepts.*

Object Data Manager (ODM) Overview in *General Programming Concepts.*

genminor Device Configuration Subroutine

Purpose

Generates either the smallest unused minor number available for a device, a preferred minor number if it is available, or a set of unused minor numbers for a device.

Syntax

long *genminor (*device_instance*, *major_no*, *preferred_minor*, *minors_in_grp*, *inc_within_grp*, *inc_btwn_grp*)

char **device_instance*;
long *major_no*;
int *preferred_minor*;
int *minors_in_grp*;
int *inc_within_grp*;
int *inc_btwn_grp*;

Parameters

device_instance	Points to a character string containing the device instance name.
major_no	Contains the major number of the device instance.
preferred_minor	Contains a single preferred minor number or a starting minor number for generating a set of numbers. In the latter case, the **genminor** subroutine can be used to get a set of minor numbers in a single call.
minors_in_grp	Indicates how many minor numbers are to be allocated.
inc_within_grp	Indicates the interval between minor numbers.
inc_btwn_grp	Indicates the interval between groups of minor numbers.

Description

The **genminor** device configuration subroutine is one of the designated routines for accessing the Customized Device Driver object class. To ensure that unique numbers are generated, the object class is locked by this routine until its completion.

If only a single preferred minor number needs to be allocated, then it should be given in the *preferred_minor* parameter. In this case, the other parameters should contain an integer value of 1. If the desired number is available, it is returned. Otherwise, a NULL pointer is returned, indicating that the requested number is in use.

If the caller has no preference and only requires one minor number, this should be indicated by passing a value of –1 in the *preferred_minor* parameter. The other parameters should all contain the integer value of 1. In this case, the **genminor** subroutine returns the lowest available minor number.

If a set of numbers is desired, then every number in the designated set must be available. An unavailable number is one which has already been assigned. To get a specific set of minor numbers allocated, the *preferred_minor* parameter contains the starting minor number. If this set has a minor number that is unavailable, then the **genminor** subroutine returns a NULL pointer indicating failure.

If the set of minor numbers needs to be allocated with the first number beginning on a particular boundary (that is, a set beginning on a multiple of 8), then the value –1 should be passed in the *preferred_minor* parameter. The *inc_btwn_grp* parameter should be set to the multiple desired. The **genminor** subroutine uses the *inc_btwn_grp* parameter to find the first complete set of available minor numbers.

If a list of minor numbers is to be returned, the return value points to the first in a list of preferred minor numbers. This pointer can then be incremented to move through the list to access each minor number. The minor numbers are returned in ascending sorted order.

File

/lib/libcfg.a

Return Values

In the case of failure, a NULL pointer is returned. If the **genminor** subroutine succeeds, a pointer to the lowest available minor number or list of minor numbers is returned.

Related Information

The **genmajor** device configuration subroutine, **getminor** device configuration subroutine, **reldevno** device configuration subroutine.

Customized Device Driver object class.

List of ODM Subroutines, Device Configuration Subroutines, Understanding Major and Minor Numbers in *Kernel Extensions and Device Support Programming Concepts.*

Object Data Manager (ODM) Overview in *General Programming Concepts.*

genseq Device Configuration Subroutine

Purpose

Generates a unique sequence number for creating a device's logical name.

Syntax

int genseq (*prefix*)
char **prefix*;

Parameter

prefix	Points to the character string containing the prefix name of the device.

Description

The **genseq** device configuration subroutine generates a unique sequence number to be concatenated with the device's prefix name. The device name in the Customized Devices object class is the concatenation of the prefix name and the sequence number.

The rules for generating sequence numbers are as follows:

1. A sequence number is a non-negative integer. The smallest sequence number is therefore 0 (zero).
2. When deriving a device instance logical name, the next available sequence number (relative to a given prefix name) is allocated. This next available sequence number is defined to be the smallest sequence number not yet allocated to device instances using the same prefix name.
3. Whether a sequence number is allocated or not is determined by the device instances in the Customized Devices object class. If an entry exists in this class using the desired prefix, then the sequence number for that entry has already been allocated.

It is up to the application to convert this sequence number to character format so that it can be concatenated to the prefix to form the device name.

File

/lib/libcfg.a

Return Values

If the **genseq** subroutine succeeds, it returns the generated sequence number in integer format. If the routine fails, it returns a value of –1.

Related Information

Customized Devices object class.

Device Configuration Subroutines, List of ODM Subroutines in *Kernel Extensions and Device Support Programming Concepts.*

Object Data Manager (ODM) Overview in *General Programming Concepts.*

getattr Device Configuration Subroutine

Purpose

Returns the current value of an attribute object or a list of current values of attribute objects from either the Customized Attribute object class or the Predefined Attribute object class.

Syntax

struct CuAt *getattr (*devname*, *attrname*, *getall*, *how_many***)**
**char ****devname*;
**char ****attrname*;
int *getall*;
**int ****how_many*;

Parameters

devname	Specifies the device logical name.
attrname	Specifies the attribute name.
getall	A Boolean flag that, when set to TRUE, indicates that a list of attributes is to be returned to the calling routine.
how_many	Points to how many attributes the **getattr** subroutine has found.

Description

The **getattr** device configuration subroutine queries the Customized Attribute object class for the attribute object matching the device logical name and the attribute name. It is the application's responsibility to lock the Device Configuration object classes.

To get a single attribute, the *getall* parameter should be set to FALSE. If the object exists in the Customized Attribute object class, a pointer to this structure is returned to the calling routine.

However, if the object is not found, then the **getattr** subroutine assumes that the attribute takes the default value found in the Predefined Attribute object class. In this case, the Predefined Attribute object class is queried for the attribute information. If this information is found, the relevant attribute values (that is, default value, flag information, and the NLS index) are copied into a Customized Attribute structure. This structure is then returned to the calling routine. Otherwise, a NULL pointer is returned indicating an error.

To get all the customized attributes for the device name, the *getall* parameter should be set to TRUE. In this case, the *attrname* parameter is ignored. The Predefined and Customized Attribute object classes are queried and a list of Customized Attribute structures is returned. The Predefined Attribute objects are copied to Customized Attribute structures so that one list may be returned.

File

/lib/libcfg.a

Return Values

Upon successful completion, the **getattr** subroutine returns a pointer to a list of Customized Attribute structures. If the operation is unsuccessful, a NULL pointer is returned.

Related Information

The **putattr** device configuration subroutine.

Predefined Attribute object class, Customized Attribute object class.

Understanding ODM Locking in *General Concepts and Procedures.*

Device Configuration Subroutines, ODM Device Configuration Object Classes: Summary in *Kernel Extensions and Device Support Programming Concepts*.

getminor Device Configuration Subroutine

Purpose

Gets the minor numbers associated with a major number from the Customized Device Driver object class.

Syntax

long *getminor (*major_no*, *how_many*, *device_instance*)
long *major_no*;
int **how_many*;
char **device_instance*;

Parameters

major_no	Specifies the major number for which the corresponding minor number or numbers is desired.
how_many	Points to the number of minor numbers found corresponding to the *major_no* parameter.
device_instance	Specifies a device instance name to use when searching for minor numbers. This parameter is used in conjunction with the *major_no* parameter.

Description

The **getminor** device configuration subroutine is one of the designated routines for accessing the Customized Device Driver object class. This subroutine queries the Customized Device Driver object class for the minor numbers associated with the given major number and/or device instance.

If the *device_instance* parameter is NULL, then only the *major_no* parameter should be used to obtain the minor numbers. Otherwise, both the *major_no* and *device_instance* parameters should be used. The number of minor numbers found in the query is returned in the *how_many* parameter.

The Customized Device Driver object class is locked exclusively by the **getminor** subroutine for the duration of the routine.

The return value pointer points to a list that contains the minor numbers associated with the major number. This pointer is then used to move through the list to access each minor number. The minor numbers are returned in ascending sorted order.

The **getminor** subroutine also returns to the calling routine, in the *how_many* parameter, the number of minor numbers in the list.

File

/lib/libcfg.a

Return Values

If the **getminor** routine fails, a NULL pointer is returned.

If the **getminor** subroutine succeeds, one of two possible values is returned. If no minor numbers are found, a NULL pointer is returned. In this case, the *how_many* parameter points to an integer value 0. However, if minor numbers are found, then a pointer to a list of minor numbers is returned. The minor numbers are returned in ascending sorted order. In the latter case, the *how_many* parameter points to the number of minor numbers found.

Related Information

The **genmajor** device configuration subroutine, **genminor** device configuration subroutine, **reldevno** device configuration subroutine.

Customized Device Driver object class.

Object Data Manager (ODM) Overview in *General Programming Concepts*.

Device Configuration Subroutines, Understanding Major and Minor Numbers in *Kernel Extensions and Device Support Programming Concepts*.

loadext Device Configuration Subroutine

Purpose

Loads or unloads kernel extensions, or queries for kernel extensions in the kernel.

Syntax

#include <sys/types.h>

mid_t loadext (*dd_name*, *load*, *query***)**
**char ****dd_name*;
int *load*;
int *query*;

Parameters

dd_name	Specifies the name of the kernel extension to be loaded, unloaded, or queried.
load	Specifies whether the **loadext** subroutine should load the kernel extension.
query	Specifies whether a query of the kernel extension should be performed.

Description

The **loadext** device configuration subroutine provides the capability to load or unload kernel extensions. It can also be used to obtain the kernel module identifier (kmid) of a previously loaded object file. The kernel extension name passed in the *dd_name* parameter can either be the base name of the object file or can contain directory path information. If the kernel extension path name supplied in the *dd_name* parameter has no leading ./ (dot, slash) or ../ (double-dot, slash) or / (slash) characters, then the **loadext** subroutine concatenates together **/etc/drivers/** and the base name passed in the *dd_name* parameter to arrive at an absolute path name. Otherwise, the path name provided in the *dd_name* parameter is used unmodified.

If the *load* parameter has a value of TRUE, then the specified kernel extension is loaded and its kernel module identifier returned. If the specified object file has already been loaded into the kernel, its load count is incremented and a new copy is not loaded.

If the *load* parameter has a value of FALSE, then the action taken depends on the value of the *query* parameter. If *query* is FALSE, then the **loadext** routine requests an unload of the specified kernel extension. This causes the kernel to decrement the load count associated with the object file. If the load count and use count of the object file become 0, the kernel unloads the object file. If the *query* parameter is TRUE, then the **loadext** subroutine queries the kernel for the kernel module ID of the specified object file. This kmid is then returned to the caller.

If both the *load* and *query* parameters have a value of TRUE, then the load function is performed.

File

/lib/libcfg.a

Return Values

Upon successful completion, the **loadext** subroutine returns the kernel module ID. Upon error or if the queried object file is not loaded, the routine returns a value of NULL.

Related Information

The **sysconfig** subroutine.

Device Configuration Subroutines, Programming in the Kernel Environment, Understanding Kernel Extension Binding in *Kernel Extensions and Device Support Programming Concepts*.

putattr Device Configuration Subroutine

Purpose

Updates attribute information in the Customized Attribute object class, creates a new object for the attribute information, or deletes an object from the Customized Attribute object class.

Syntax

int putattr (*cuobj*)
struct CuAt **cuobj*;

Parameter

cuobj	Specifies the attribute object.

Description

The **putattr** device configuration subroutine either updates an old attribute object, creates a new one in the Customized Attribute object class, or deletes an existing object. The subroutine queries the Customized Attribute object class to determine whether an object already exists with the device name and attribute name specified in the *cuobj* parameter.

If the attribute is found in the Customized Attribute object class and its value (as given in the *cuobj* parameter) is to be changed back to the default value for this attribute, the customized object is deleted. Otherwise, the customized object is simply updated.

If the attribute object does not already exist and its attribute value is being changed to a non-default value, a new object is added to the Customized Attribute object class with the information given in the *cuobj* parameter.

File

/lib/libcfg.a

Return Values

0	Indicates a successful operation.
–1	Indicates a failed operation.

Related Information

The **getattr** device configuration library routine.

Customized Attribute object class.

Object Data Manager (ODM) Overview in *General Programming Concepts.*

Device Configuration Subroutines in *Kernel Extensions and Device Support Programming Concepts.*

reldevno Device Configuration Subroutine

Purpose

Releases the minor number or major number, or both, for a device instance.

Syntax

int
reldevno (*device_instance_name*, *release***)**
**char ****device_instance_name*;
int *release*;

Parameters

device_instance_name	Points to the character string containing the device instance name.
release	Specifies whether the major number should be released. A value of TRUE releases the major number. A value of FALSE does not release the major number.

Description

The **reldevno** device configuration subroutine is one of the designated access routines to the Customized Device Driver object class. This object class is locked exclusively by this routine until its completion. All minor numbers associated with the device instance name are deleted from the Customized Device Driver object class. That is, each object is deleted from the class. This releases the minor numbers for re-use.

The major number is released for re-use if the following two conditions exist:

- The object to be deleted contains the last minor number for a major number.
- The *release* parameter is set to TRUE.

If you would rather explicitly release the major number yourself, then the **relmajor** device configuration subroutine can be called. In this case, you should also set the *release* parameter to FALSE. All special files, including symbolically linked special files, corresponding to the deleted objects are deleted from the file system.

File

/lib/libcfg.a

Return Values

0	Indicates successful completion.
–1	Indicates a failure to release the minor number and major number, or both.

Related Information

The **genmajor** device configuration subroutine, **genminor** device configuration subroutine, **relmajor** device configuration subroutine.

Customized Device Driver object class.

Special File Overview in *Files Reference.*

Object Data Manager (ODM) Overview in *General Programming Concepts.*

Device Configuration Subroutines, Understanding Major and Minor Numbers in *Kernel Extensions and Device Support Programming Concepts.*

relmajor Device Configuration Subroutine

Purpose

Releases the major number associated with the specified device driver instance name.

Syntax

int relmajor (*device_driver_instance_name***)**
**char ***device_driver_instance_name*;

Parameter

device_driver_instance_name	Points to a character string containing the device driver instance name.

Description

The **relmajor** device configuration subroutine is one of the designated access routines to the Customized Device Driver object class. To ensure that unique major numbers are generated, the Customized Device Driver object class is locked exclusively by this routine until the major number has been released.

The **relmajor** routine deletes the object containing the major number for the device driver instance name.

File

/lib/libcfg.a

Return Values

0	Indicates successful completion.
–1	Indicates a failure to release the major number.

Related Information

The **genmajor** device configuration subroutine, **reldevno** device configuration subroutine.

Customized Device Driver object class.

Object Data Manager (ODM) Overview in *General Programming Concepts.*

Device Configuration Subroutines, Understanding Major and Minor Numbers in *Kernel Extensions and Device Support Programming Concepts.*

relseq Device Configuration Subroutine

Purpose

Releases the unique sequence number associated with the device's logical name.

Syntax

int relseq (*prefix*, *seqno*)
char **prefix*;
int *seqno*;

Parameters

prefix	Specifies a pointer to the character string containing the prefix name.
seqno	Specifies the sequence number.

Description

The **relseq** device configuration subroutine releases the sequence number associated with the prefix name. The Customized Device Driver object class is locked exclusively by the **relseq** subroutine until its completion. The object containing the sequence number is then deleted from the Customized Device Driver object class.

File

/lib/libcfg.a

Return Values

0	Indicates a successful completion.
–1	Indicates a failure to release the sequence number.

Related Information

The **genseq** device configuration subroutine.

Customized Device Driver object class.

Device Configuration Subroutines in *Kernel Extensions and Device Support Programming Concepts.*

Object Data Manager (ODM) Overview in *General Programming Concepts.*

ODM Device Configuration Object Classes

The following is a list of the Device Configuration Database object classes:

Config_Rules	Configuration Rules
CuAt	Customized Attribute
CuDep	Customized Dependency
CuDv	Customized Devices
CuDvDr	Customized Device Driver
CuVPD	Customized Vital Product Data.
PdAt	Predefined Attribute
PdCn	Predefined Connection
PdDv	Predefined Devices

Related Information

Device Configuration Subsystem: Programming Introduction, Writing a Device Method Overview in *Kernel Extensions and Device Support Programming Concepts.*

Object Data Manager (ODM) Overview in *General Programming Concepts.*

Config_Rules Object Class (Configuration Rules)

The Configuration Rules (Config_Rules) object class contains the configuration rules used by the Configuration Manager. The Configuration Manager runs in two phases during system boot. The first phase is responsible for configuring the base devices so that the real root device can be configured and made ready for operation. The second phase configures the rest of the devices in the system after the root file system is up and running. The Configuration Manager can also be invoked at runtime. The Configuration Manager routine is driven by the rules in the Config_Rules object class.

The Config_Rules object class is preloaded with predefined configuration rules when the system is delivered. You can use the ODM Object Editor to add, remove, change, and show new or existing configuration rules in this object class to customize the behavior of the Configuration Manager. However, any changes to a rule must be written to the boot file system to be effective. This is done with the **bosboot** command.

All logical and physical devices in the system are organized in clusters of tree structures called nodes. For information on nodes or tree structures, see the Device Configuration Manager Overview. The rules in the Config_Rules object class specify program names that the Configuration Manager executes. Usually, these programs are the configuration programs for the top of the nodes. When these programs are invoked, the names of the next lower-level devices that need to be configured are returned in standard output.

The Configuration Manager configures the next lower-level devices by invoking the Configure method for those devices. In turn, those devices return a list of device names to be configured. This process is repeated until no more device names are returned. All devices in the same node are configured in a transverse order. There are three types of rules: phase 1, phase 2, and service.

The second phase of system boot requires two sets of rules: phase 2 and service. The position of the key on the front panel determines which set of rules is used. The service rules are used when the key is in the service position. If the key is in any other position, the phase 2 rules are used. Different modes of rules are indicated in the Configuration Manager Phase descriptor of this object class.

Configuration Rules Object Class

The Config_Rules object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_SHORT	phase	Configuration manager phase	Required
ODM_SHORT	seq	Sequence value	Required
ODM_VCHAR	rule_value[RULESIZE]	Rule value	Required

These fields are described as follows:

Cfgmgr Phase This field indicates which phase a rule should be executed under: Phase 1, Phase 2, or Phase 2 service.

1 Indicates that the rule should be executed in Phase 1.

2 Indicates that the rule should be executed in Phase 2.

3 Indicates that the rule should be executed in Phase 2 service mode.

Sequence Value
In relation to the other rules of this phase, **seq** indicates the order in which to execute this program. In general, the lower the **seq** number, the higher the priority. For example, a **rule** with a **seq** number of 2 is executed before a **rule** with a **seq** number of 5. There is one exception to this: a value of 0 indicates a DON'T_CARE condition, and any **rule** with a **seq** number of 0 will be executed last.

Rule Value This is the full path name of the program to be invoked. The Rule Value descriptor may also contain any options that should be passed to that program. However, options must follow the program name, as the whole string will be executed as if it has been typed in on the command line. Note that there is one rule for each program to execute. If multiple programs are needed, then multiple rules must be added.

Rule Values		
Phase	**Sequence**	**Rule Value**
1	1	**/etc/methods/defsys**
1	5	**/etc/methods/deflvm**
2	1	**/etc/methods/defsys**
2	5	**/etc/methods/ptynode**
2	10	**/etc/methods/starthft**
2	15	**/etc/methods/starttty**
2	20	**/etc/methods/netstart.sh**
3	1	**/etc/methods/defsys**
3	5	**/etc/methods/ptynode**
3	10	**/etc/methods/starthft**
3	15	**/etc/methods/starttty**

Related Information

The **bosboot** command.

Writing a Configure Method, Writing A Device Method.

The Device Configuration Manager Overview, Understanding System Boot Processing, Device Configuration Subsystem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Object Data Manager (ODM) Overview, Understanding ODM Object Classes and Objects, Understanding ODM Descriptors in *General Programming Concepts.*

CuAt Object Class (Customized Attribute)

The Customized Attribute (CuAt) object class contains customized device-specific attribute information.

Device instances represented in the Customized Devices (CuDv) object class have attributes found in either the Predefined Attribute (PdAt) object class or the CuAt object class. There is an entry in the CuAt object class for attributes that take non-default values. Attributes taking the default value are found in the PdAt object class. Each entry describes the current value of the attribute.

When changing the value of an attribute, the Predefined Attribute object class must be referenced to determine other possible attribute values.

Both attribute object classes must be queried to get a complete set of current values for a particular device's attributes. Use the **getattr** and **putattr** routines to retrieve or modify customized attributes.

Customized Attribute Object Class

The Customized Attribute object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	name[NAMESIZE]	Device name	Required
ODM_CHAR	attribute[ATTRNAMESIZE]	Attribute name	Required
ODM_VCHAR	value[ATTRVALSIZE]	Attribute value	Required
ODM_CHAR	type[FLAGSIZE]	Attribute type	Required
ODM_CHAR	generic[FLAGSIZE]	Generic attribute flags	Optional
ODM_CHAR	rep[FLAGSIZE]	Attribute representation flags	Required
ODM_SHORT	nls_index	NLS index	Optional

These fields are described as follows:

Device Name Identifies the logical name of the device instance to which this attribute is associated.

Attribute Name Identifies the name of a customized device attribute.

Attribute Value Identifies a customized value associated with the corresponding Attribute Name. This value is a non-default value.

Attribute Type Identifies the Attribute Type associated with the Attribute Name. This field is copied from the Attribute Type descriptor in the corresponding PdAt object when the CuAt object is created.

Generic Attribute Flags
Identifies the Generic Attribute flag or flags associated with the Attribute Name. This field is copied from the Generic Attribute flags descriptor in the corresponding PdAt object when the CuAt object is created.

Attribute Representation Flags	Identifies the Attribute Value's representation. This field is copied from the Attribute Representation flags descriptor in the corresponding PdAt object when the CuAt object is created.
NLS Index	Identifies the message number in the NLS message catalogue that contains a textual description of the attribute. This field is copied from the NLS Index descriptor in the corresponding PdAt object when the CuAt object is created.

Related Information

The **getattr** device configuration subroutine, **putattr** device configuration subroutine.

Customized Devices object class, Predefined Attribute object class.

Object Data Manager (ODM) Overview, Understanding ODM Object Classes and Objects, Understanding ODM Descriptors in *General Programming Concepts.*

Device Configuration Subroutines in *Kernel Extensions and Device Support Programming Concepts.*

CuDep Object Class (Customized Dependency)

The Customized Dependency (CuDep) object class describes device instances that depend on other device instances. Dependency does not imply a physical connection. This object class describes the dependence links between logical devices and physical devices as well as dependence links between logical devices. Physical dependencies of one device on another device are recorded in the Customized Device (CuDev) object class.

The Devices Graph diagram demonstrates instances of dependency and connection between devices.

Customized Dependency Object Class

The CuDep object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	name[NAMESIZE]	Device name	Required
ODM_CHAR	dependency[NAMESIZE]	Dependency (device logical name)	Required

These descriptors are defined as follows:

Device Name	Identifies the logical name of the device having a dependency.
Dependency	Identifies the logical name of the device instance on which there is a dependency. For example, a mouse, keyboard, and display might all be dependencies of a device instance of `hft0`.

Related Information

Customized Device object class.

Object Data Manager (ODM) Overview, Understanding ODM Object Classes and Objects, Understanding ODM Descriptors in *General Programming Concepts.*

Device Configuration Manager Overview in *Kernel Extensions and Device Support Programming Concepts.*

CuDv Object Class (Customized Devices)

The Customized Devices (CuDv) object class contains entries for all device instances defined in the system. As the name implies, a defined device object is an object that a Define method has created in the CuDv object class. A defined device instance may or may not have a corresponding actual device attached to the system.

A CuDv object contains attributes and connections specific to the device instance. Each device instance, distinguished by a unique logical name, is represented by an object in the CuDv object class. The Customized database is updated twice, during system boot and at runtime, to define new devices, remove undefined devices, or update the information for a device whose attributes have been changed.

Customized Devices Object Class

The CuDv object class contains the following fields:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	name[NAMESIZE]	Device name	Required
ODM_SHORT	status	Device status flag	Required
ODM_SHORT	chgstatus	Change status flag	Required
ODM_CHAR	ddins[TYPESIZE]	Device driver instance	Optional
ODM_CHAR	location[LOCSIZE]	Location code	Optional
ODM_CHAR	parent[NAMESIZE]	Parent device logical name	Optional
ODM_CHAR	connwhere[LOCSIZE]	Location where device is connected	Optional
ODM_LINK	PdDvLn	LINK to Predefined Devices object class	Required

These fields have the following descriptions:

Device Name A Customized Device object for a device instance is assigned a unique logical name to distinguish the instance from other device instances. The device logical name of a device instance is derived during Define method processing. The rules for deriving a device logical name are:

1. The name should start with a *prefix name* pre-assigned to the device instance's associated device type. The prefix name can be retrieved from the Prefix Name descriptor in the Predefined Device object associated with the device type.
2. To complete the logical device name, a *sequence number* is usually appended to the prefix name. This sequence number is unique among all defined device instances using the same prefix name. Use the following subrules when generating sequence numbers:
    a. A sequence number is a non-negative integer represented in character format. Therefore, the smallest available sequence number is 0 (zero).

b. The next available sequence number relative to a given prefix name should be allocated when deriving a device instance logical name.

c. The next available sequence number relative to a given prefix name is defined to be the smallest sequence number not yet allocated to defined device instances using the same prefix name.

   For example, if `tty0`, `tty1`, `tty3`, `tty5` and `tty6` are currently assigned to defined device instances, then the next available sequence number for a device instance with the `tty` prefix name is `2`. This results in a logical device name of `tty2`.

   The **genseq** subroutine can be used by a Define method to obtain the next available sequence number.

Device Status Flag

Identifies the current status of the device instance. The device methods are responsible for setting the Device Status flags of device instances. When the Define method defines a device instance, the device's device status is set to `defined`. When the Configure method configures a device instance, the device's device status is typically set to `available`. The Configure method takes a device to the Stopped state only if the device supports the Stopped state.

When the Start method starts a device instance, its device status is changed from the Stopped state to the Available state. Applying a Stop method on a started device instance changes the device status from the Available state to the Stopped state. Applying an Unconfigure method on a configured device instance changes the device status from the Available state to the Defined state. If the device supports the Stopped state, the Unconfigure method takes the device from the Stopped state to the Defined state.

Understanding Device States provides more information about the Available, Defined, and Stopped states.

The possible status values are:

DEFINED	Identifies a device instance in the Defined state.
AVAILABLE	Identifies a device instance in the Available state.
STOPPED	Identifies a device instance in the Stopped state.

Change Status Flag

This flag tells whether the device instance has been altered since the last system boot. The diagnostics facility uses this flag to validate system configuration. The flag can take on these values:

NEW	Specifies whether the device instance is new to the current system boot.
DONT_CARE	Identifies the device as one whose presence or uniqueness cannot be determined. For these devices, the new, same, and missing states have no meaning.
SAME	Specifies whether the device instance was known to the system prior to the current system boot.
MISSING	Specifies whether the device instance is missing. This is true if the device is in the CuDv object class, but is not physically present.

Device Driver Instance
This field typically contains the same value as the Device Driver Name descriptor in the Predefined Devices (PdDv) object class if the device driver supports only one major number. For a driver that uses multiple major numbers (for example, the logical volume device driver), unique instance names must be generated for each major number. Since the logical volume uses a different major number for each volume group, the volume group logical names would serve this purpose. This field is filled in with a null string if the device instance does not have a corresponding device driver.

Location Code Identifies the location code of the device. This field provides a means of identifying physical devices. The location code format is defined as **AB-CD-EF-GH** where:

AB	Is the drawer ID used to identify the CPU and Async drawers.
CD	Is the slot ID used to identify the location of an adapter, memory card, or SLA (Serial Link Adapter).
EF	Is the connector ID used to identify the adapter connector that something is attached to.
GH	Is the port or device or FRU ID used to identify a port, device, or FRU, respectively.

For more information on the location code format, see Understanding Location Codes in Devices Overview for System Management.

Parent Device Logical Name
Identifies the logical name of the parent device instance. In the case of a real device, this indicates the logical name of the parent device to which this device is connected. More generally, the specified parent device is the device whose Configure method is responsible for returning the logical name of this device to the Configuration Manager for configuring this device. This field is filled in with a null string for a node device.

Location Where Device Is Connected
Identifies the specific location on the parent device instance where this device is connected. The term *location* is used in a generic sense. For some device instances such as the AIX bus, *location* indicates a slot on the bus. For device instances such as the SCSI adapter, the term indicates a logical port (that is, a SCSI ID and Logical Unit Number combination).

For example, for a bus device, the location can refer to a specific slot on the bus, with values 1, 2, 3 .... For a multiport serial adapter device, the location can refer to a specific port on the adapter, with values 0, 1, ...

LINK to Predefined Devices Object Class (PdDvLn)
Provides a link to the device instance's predefined information through the Unique Type descriptor in the PdDv object class.

Related Information

The Define device configuration method, Configure configuration method, Change configuration method, Undefine configuration method, Unconfigure configuration method, Start and Stop configuration method.

Predefined Devices (PdDv) object class.

The **genseq** subroutine.

The SCSI Adapter Device Driver, Physical Volumes and the Logical Volume Device Driver.

Devices Overview for System Management, Object Data Manager (ODM) Overview, Understanding Location Codes, Understanding ODM Object Classes and Objects, Understanding ODM Descriptors in *General Programming Concepts*.

Understanding Major and Minor Numbers for a Special File, Understanding Device States, Device Configuration Subsystem: Programming Introduction, Configuration Manager Overview in *Kernel Extensions and Device Support Programming Concepts*.

CuDvDr Object Class (Customized Device Driver)

The Customized Device Driver (CuDvDr) object class stores information about critical resources that need concurrency management through the use of the Device Configuration Library routines. You should only access this object class through these five Device Configuration Library routines: the **genmajor**, **genminor**, **relmajor**, **reldevno**, and **getminor** routines.

These routines exclusively lock this class so that accesses to it are serialized. The **genmajor** and **genminor** routines return the major and minor number to the calling method. Similarly, the **reldevno** or **relmajor** routine releases the major or minor number from this object class.

Customized Device Driver Object Class

The CuDvDr object class contains the following fields:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	resource[RESOURCESIZE]	Resource Name	Required
ODM_CHAR	value1[VALUESIZE]	Value1	Required
ODM_CHAR	value2[VALUESIZE]	Value2	Required
ODM_CHAR	value3[VALUESIZE]	Value3	Required

The Resource descriptor determines the nature of the values in the Value1, Value2, and Value3 descriptors. Possible values for the Resource descriptor are the strings **devno** and **ddins**.

The following table specifies the contents of the Value1, Value2, and Value3 descriptors, depending on the contents of the Resource descriptor.

Resource	Value1	Value2	Value3
devno	Major number	Minor number	Device instance name
ddins	Dd instance name	Major number	Null string

When the resource field contains the **devno** string, the Value1 field contains the device major number, Value2 the device minor number, and Value3 the device instance name. These value fields are filled in by the **genminor** subroutine, which takes a major number and device instance name as input, and generates the minor number and resulting **devno** Customized Device Driver object.

When the resource field contains the **ddins** string, the Value1 field contains the device driver instance name. This is typically the device driver name obtained from the Device Driver Name descriptor of the Predefined Device object. However, this name can be any unique string and is used by device methods to obtain the device driver major number. The Value2 field contains the device major number and the Value3 field is not used. These value fields are set by the **genmajor** subroutine, which takes a device instance name as input, and generates the corresponding major number, and resulting **ddins** Customized Device Driver object.

Related Information

Predefined Device object class.

The **genmajor** device configuration subroutine, **genminor** device configuration subroutine, **relmajor** device configuration subroutine, **reldevno** device configuration subroutine, and **getminor** device configuration subroutine.

Object Data Manager (ODM) Overview, Understanding ODM Object Classes and Objects, Understanding ODM Descriptors in *General Programming Concepts.*

Device Driver Introduction, List of Device Configuration Subroutines, Understanding Major and Minor Numbers for a Special File in *Kernel Extensions and Device Support Programming Concepts*.

CuVPD Object Class (Customized VPD)

Description

The Customized VPD (CuVPD) object class contains the Vital Product Data (VPD) for customized devices. VPD can be either machine-readable VPD or manually-entered user VPD information.

The CuVPD object class contains the following descriptors:

Customized Vital Product Data Object Class

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	name[NAMESIZE]	Device name	Required
ODM_SHORT	vpd_type	VPD type	Required
ODM_LONGCHAR	vpd[VPDSIZE]	VPD	Required

These fields are described as follows:

Device Name Identifies the device logical name to which this VPD information belongs.

VPD Type Identifies the VPD as either machine-readable or manually entered. The possible values:

HW_VPD	Identifies machine-readable VPD.
USER_VPD	Identifies manually entered VPD.

VPD Identifies the Vital Product Data for the device. For machine-readable VPD, an entry in this field might include such information as serial numbers, engineering change levels, and part numbers.

Manually-entered VPD is intended for accounting purposes. For example, the user may want the name of the individual responsible for the device as well as his or her office number.

Related Information

Object Data Manager (ODM) Overview, Understanding ODM Object Classes and Objects, Understanding ODM Descriptors in *General Programming Concepts*.

The *RISC System/6000 Hardware Technical Reference* provides more details on the VPD.

PdAt Object Class (Predefined Attribute)

The Predefined Attribute (PdAt) object class contains an entry for each existing attribute for each device represented in the Predefined Devices object class. An attribute, in this sense, is any device-dependent information not represented in the PdDv object class. This includes information such as interrupt levels, bus I/O address ranges, baud rates, parity settings, block sizes, and microcode file names.

Each object in this object class represents a particular attribute belonging to a particular device class-subclass-type. Each object contains the attribute name, default value, list or range of all possible values, width, flags, and an NLS description. The flags provide further information to describe an attribute.

Note: For a device being defined or configured, only the attributes that take a nondefault value are copied into the Customized Attribute (CuAt) object class. In other words, for a device being customized, if its attribute value is the default value in the PdDv object class, then there will not be an entry for the attribute in the CuAt object class.

Types of Attributes

There are three types of attributes. Most are *regular* attributes, which typically describe a specific attribute of a device. The *group* attribute type provides a grouping of regular attributes. The *shared* attribute type identifies devices that must all share the given attribute.

A *shared* attribute identifies another regular attribute as one that must be shared. This attribute is always a bus resource. Other regular attributes (for example, bus interrupt levels) can be shared by devices but are not themselves *shared* attributes. *Shared* attributes *require* that the relevant devices have the same values for this attribute. The **Attribute Value** descriptor for the shared attribute gives the name of the regular attribute that must be shared.

A *group* attribute specifies a set of other attributes whose values are chosen as a group, as well as a group attribute number used to choose the default values. Each attribute listed within a group has an associated list of possible values it can take. These values must be represented as a list, not as a range. For each attribute within the group, the list of possible values must also have the same number of choices. For example, if the possible number of values is *n*, the group attribute number itself can range from 0 to *n*–1. The particular value chosen for the group indicates the value to pick for each of the attributes in the group. For example, if the group attribute number is 0, then the value for each of the attributes in the group is the first value from their respective lists.

Predefined Attribute Object Class Descriptors

The Predefined Attribute object class contains the following fields:

Predefined Attribute Object Class Fields			**Part 1 of 2**
ODM Type	**Descriptor Name**	**Description**	**Status**
ODM_CHAR	uniquetype[UNIQUESIZE]	Unique type	Required
ODM_CHAR	attribute[ATTRNAMESIZE]	Attribute name	Required
ODM_VCHAR	deflt[DEFAULTSIZE]	Default value	Required
ODM_VCHAR	values[ATTRVALSIZE]	Attribute values	Required

Predefined Attribute Object Class Fields			Part 2 of 2
ODM Type	**Descriptor Name**	**Description**	**Status**
ODM_CHAR	width[WIDTHSIZE]	Width	Optional
ODM_CHAR	type[FLAGSIZE]	Attribute type flags	Required
ODM_CHAR	generic[FLAGSIZE]	Generic attribute flags	Optional
ODM_CHAR	rep[FLAGSIZE]	Attribute representation flags	Required
ODM_SHORT	nls_index	NLS index	Optional

These fields are described as follows:

Unique Type Identifies the class-subclass-type name of the device to which this attribute is associated. This descriptor is the same as the Unique Type descriptor in the PdDv object class.

Attribute Name Identifies the name of the device attribute. This is the name that can be passed to the **mkdev** and **chdev** configuration commands and device methods in the attribute-name and attribute-value pairs.

Default Value If there are several choices or even if there is only one choice for the attribute value, the default is the value that the attribute is normally set to. For groups, the default value is the group attribute number. For example, if the possible number of choices in a group is *n*, the group attribute number is a number between 0 and *n*–1. For shared attributes, the default value is set to a null string.

When a device is defined in the system, attributes that take nondefault values are found in the CuAt object class. Attributes that take the default value are found in this object class. Attributes that take on the default value are not copied over to the CuAt object class. Therefore, both attribute object classes must be queried to get a complete set of customized attributes for a particular device.

Possible Values Identifies the possible values that can be associated with the attribute name. The format of the value is determined by the Attribute Representation flags. For regular attributes, the possible values can be represented as a string, hexadecimal, octal, or decimal. In addition, they can be represented as a range or an enumerated list. If there is only one possible value, then the value can be represented either as a single value or as an enumerated list with one entry. The latter is recommended, since the use of enumerated lists allows the **attrval** subroutine to check that a given value is in fact a possible values.

If the value is hexadecimal, then it is prefixed with the 0x notation. If the value is octal, the value is prefixed with a leading zero. If the value is decimal, the value is its significant digits. If the value is a string, the string itself should not have embedded commas since commas are used as separators of items in an enumerated list.

A range is represented as a triplet of values: *lowerlimit-upperlimit, increment value.* The *lowerlimit* variable indicates the value of the first possible choice. The *upperlimit* variable indicates the value of the last possible choice. The *lowerlimit* and *upperlimit* values are separated by a - (hyphen). Values between the *lowerlimit* and *upperlimit* values are obtained by adding multiples of the *increment value* variable to the *lowerlimit* variable. The *upperlimit* and *increment value* variables are separated by a , (comma).

Only numeric values are used for ranges. Also, discontinuous ranges (for example, 1 to 3, 6 to 8) are disallowed. A combination of list and ranges is not allowed.

An enumerated list contains values that are comma-separated.

If the attribute is a group, the Possible Values descriptor contains a list of attributes composing the group, separated by commas.

If the attribute is shared, the Possible Values descriptor contains the name of the bus resource regular attribute that must be shared with another device.

Width If the attribute is a regular attribute, the Width descriptor identifies the amount of resource used by the attribute. For example, if the attribute indicates the starting bus memory address for an adapter card, this field indicates the range of bus memory that must be allocated to the adapter. Width only applies to attributes with the M (bus memory address) and the O (bus I/O address) Attribute Types. For all other attributes, a null string is used to fill in this field.

Attribute Type Identifies the attribute type. Only one Attribute Type must be specified. The characters A, M, I, O, and P represent bus resources that are regular attributes.

For regular attributes, the following Attribute Types are defined:

R	Indicates a regular attribute that is not a bus resource.

The following are the bus resources types for regular attributes:

A	Indicates DMA arbitration level.
M	Indicates bus memory address.
I	Indicates bus interrupt level.
O	Indicates bus I/O address.
P	Indicates priority class.

For non-regular attributes, the following Attribute Types are defined:

G	Indicates a group.
S	Indicates a shared attribute.

Generic Attribute Flags
Identifies the flags that can apply to any regular attribute. Any combination, one, both, or none, of these flags is valid. This descriptor should be a null string for group and shared attributes.

These are the defined Generic Attribute flags:

D	Indicates a displayable attribute. The **lsattr** command displays only attributes with this flag.
U	Indicates an attribute whose value can be set by the user.

Attribute Representation Flags
Indicates the representation of the regular attribute values. For group and shared attributes, which have no associated attribute representation, this descriptor is set to a null string. Either the **n** or **s** flag, both of which indicate value representation, must be specified.

The **r** and **l** flags indicate, respectively, a range and an enumerated list, and are optional. If neither **r** nor **l** is specified, then the **attrval** subroutine will not verify that the value falls within the range or the list.

These are the defined Attribute Representation flags:

n	Indicates that the attribute value is numeric, either decimal, hex, or octal.
s	Indicates that the attribute value is a character string.
r	Indicates that the attribute value is a range of the form: *lowerlimit–upperlimit,increment value.*
l	Indicates that the attribute value is an enumerated list of values.

NLS Index
Identifies the message number in the NLS message catalog of the message containing a textual description of the attribute. Only displayable attributes, as identified by the Generic Attribute flags descriptor, need an NLS message. If the attribute is not displayable, the NLS Index can be set to a value of 0. The catalog file name and the set number associated with the message number are stored in the PdDv object class.

Related Information

The **lsattr** command, **mkdev** command, **chdev** command.

Predefined Devices object class, Customized Attribute object class.

The **attrval** subroutine.

Writing A Device Method.

Object Data Manager (ODM) Overview, Understanding ODM Object Classes and Objects, Understanding ODM Descriptors in *General Programming Concepts.*

Device Configuration Subsystem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Adapter-Specific Considerations for the PdAt Object Class

The various bus resources required by an adapter card are represented as attributes in the Predefined Attribute (PdAt) object class. The current values assigned, if different from the default values, are represented in the Customized Attribute (CuAt) object class just like all other device attributes. To assign bus resources, the Bus Configurator obtains the bus resource attributes for an adapter from both the PdAt and CuAt object classes. It also updates the CuAt object class, as necessary, to resolve any bus resource conflicts.

The following additional guidelines apply to bus resource attributes:

The Attribute Type descriptor must indicate the type of bus resource. The values are as follows:

A	Indicates DMA arbitration level.
M	Indicates bus memory address.
I	Indicates bus interrupt level.
O	Indicates bus I/O address.
P	Indicates interrupt priority class.
G	Indicates a group.
S	Indicates an attribute that must be *shared* with another adapter.

For bus memory and bus I/O addresses, the Width descriptor must identify the amount of address space to be assigned. The Width descriptor for all other attributes should be set to a null string.

The last two attribute types, G and S, are special-purpose types that the Bus Configurator recognizes. If an adapter has resources whose values cannot be assigned independently of each other, a *group* attribute will identify them to the Bus Configurator. For example, an adapter card might have an interrupt level that depends on the bus memory address assigned. Suppose that interrupt level 3 must be used with bus memory address 0x1000000, while interrupt level 4 must be used with bus memory address 0x2000000. This relationship can be described using the group attribute as discussed in PdAt Object Class.

Occasionally, all cards of a particular type or types must use the same bus resource when present in the system. This is especially true of interrupt levels. Although most adapter's resources can be assigned completely independent of other adapters, even those of the same type, it is not uncommon to find adapters that need to be tied together. An adapter card having a bus resource that must be shared with another adapter needs an attribute of type S to describe the relationship.

PdAt Descriptors for Type S Attributes

The PdAt descriptors for an attribute of type S should be set as follows:

Unique Type	Indicates the unique type of the adapter.
Attribute Name	Specifies the name assigned to this attribute.
Default Value	Set to a null string.
Possible Values	Contains the name of the attribute that must be shared with another adapter or adapters.
Width	Set to a null string.
Attribute Type	Set to S.

Generic Attribute Flags
Set to a null string. This attribute must neither be displayed nor set by the user.

Attribute Representation Flags
Set to `sl`, indicating an enumerated list of strings, even though the list consists of only one item.

NLS Index Set to `0` since the attribute is not displayable.

The type S attribute identifies a bus resource attribute that must be shared. The other adapters are identifiable by attributes of type S with the same attribute name. The attribute name for the type S attribute serves as a key to identify all the adapters.

For an example, suppose an adapter with unique type `adapter/mca/X` must share its interrupt level with an adapter of unique type `adapter/mca/Y`. The following attributes describe such a relationship:

The attributes for `X`'s interrupt level:

- Attribute Name = int_level
- Default Value = 3
- Possible Values = 2 –9,1
- Width = null string
- Unique Type = adapter/mca/X
- Attribute Type = I
- Generic Attribute Flags = D (displayable but not user-setable)
- Attribute Representation Flags = nr
- NLS Index = 12 (message number for text description).

The attribute describing `X`'s *shared* interrupt level:

- Unique Type = adapter/mca/X
- Attribute Name = shared_intr
- Default Value = null string
- Possible Values = “int_level”
- Width = null string
- Attribute Type = S
- Generic Attribute Flags = null string
- Attribute Representation Flags = sl
- NLS Index = 0.

The attribute for `Y`'s interrupt level:

- Unique Type = adapter/mca/Y
- Attribute Name = interrupt
- Default Value = 7
- Possible Values = 2,3,4,5,7,9
- Width = null string
- Attribute Type = I
- Generic Attribute Flags = D (displayable but not user-setable)
- Attribute Representation Flags = nl
- NLS Index = 6 (message number for text description).

The attribute describing `Y`'s *shared* interrupt level:

- Unique Type = adapter/mca/Y
- Attribute Name = shared_intr
- Default Value = null string
- Possible Values = "interrupt"
- Width = null string
- Attribute Type = S
- Generic Attribute Flags = null string
- Attribute Representation Flags = sl
- NLS Index = 0.

Note that the two adapters have quite different attributes describing their interrupt levels. The attribute name is also different. However, their attributes describing what must be shared have the same name: `shared_intr`.

Adapter bus resource attributes can be displayed, but not set, by the user. In other words, the Generic Attribute Flags descriptor can either be a null string or the character `D`, but cannot be `U` or `DU`. The Bus Configurator has total control over the assignment of bus resources. These resources cannot be changed to user-supplied values by the Change method.

Note: This does not apply to any other attribute the adapter may have with attribute type R.

Related Information

Writing a Change Method.

Customized Attributes (CuAt) Cbject Class, Predefined Attribute (PdAt) Object Class.

Adapter-Specific Considerations for the Predefined Devices (PdDv) Object Class.

PdCn Object Class (Predefined Connection)

The Predefined Connection (PdCn) object class contains connection information for intermediate devices. This object class also includes predefined dependency information. For each connection location, there are one or more objects describing the subclasses of devices that can be connected. This information is useful, for example, in verifying whether a device instance to be defined and configured can be connected to a given device.

Predefined Connection Object Class

The PdCn object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	uniquetype[UNIQUESIZE]	Unique type	Required
ODM_CHAR	connkey[KEYSIZE]	Connection key	Required
ODM_CHAR	connwhere[LOCSIZE]	Connection location	Required

These fields are described as follows:

Unique Type Identifies the intermediate device's class-subclass-type name. For a device with dependency information, this descriptor identifies the unique type of the device on which there is a dependency. This descriptor contains the same information as in the Unique Type descriptor in the Predefined Devices (PdDv) object class.

Connection Key
Identifies a subclass of devices that can connect to the intermediate device at the specified location. For a device with dependency information, this descriptor serves to identify the device indicated by the Unique Type field to the devices that depend on it.

Connection Location
Identifies a specific location on the intermediate device where a child device can be connected. For a device with dependency information, this descriptor is not always required and consequently may be filled in with a null string.

The term *location* is used in a generic sense. For example, for a bus device, the location can refer to a specific slot on the bus, with values 1, 2, 3 ... For a multiport serial adapter device, the location can refer to a specific port on the adapter with values 0, 1, ...

Related Information

Predefined Devices (PdDv) object class.

Object Data Manager (ODM) Overview, Understanding ODM Object Classes and Objects, Understanding ODM Descriptors in *General Programming Concepts.*

Device Configuration Subsystem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Predefined Devices (PdDv) Object Class

The Predefined Devices (PdDv) object class contains entries for all known device types supported by the system. The term *devices* is used in the general sense in this context. Devices include intermediate devices (for example, adapters) and terminal devices (for example, disks, printers, display terminals, and keyboards). Pseudo-devices, including pseudo terminals, logical volumes, and TCP/IP, are also included under devices. Pseudo-devices can either be intermediate or terminal devices.

Each device type, as determined by class-subclass-type information, is represented by an object in the PdDv object class. These objects contain basic information about the devices, such as device method names and how to access information contained in other object classes. The PdDv object class is referenced by the Customized Devices (CuDv) object class by a link that keys into the Unique Type descriptor. This descriptor is uniquely identified by the class-subclass-type information.

Typically, the Predefined database is consulted but never modified during system boot or runtime. One exception occurs when a new device is to be added to the Predefined database. In this case, the predefined information for the new device must be added into the Predefined database.

You build a Predefined Device object by defining the objects in a file in stanza format and then processing the file with the **odmadd** command or the **odm_add_obj** subroutine. See the **odmadd** command or the **odm_add_obj** subroutine for information on creating the input file and compiling the object definitions into objects.

Note: When coding an object in this object class, set unused empty strings to "" (two double quotation marks with no separating space) and unused integer fields to 0 (zero).

Predefined Devices Object Class

Each Predefined Devices object corresponds to an instance of the PdDv object class. The descriptors for the PdDv object class are:

Predefined Devices Object Class Fields			**Part 1 of 2**
ODM Type	**Descriptor Name**	**Description**	**Descriptor Status**
ODM_CHAR	type[TYPESIZE]	Device type	Required
ODM_CHAR	class[CLASSIZE]	Device class	Required
ODM_CHAR	subclass[CLASSIZE]	Device subclass	Required
ODM_CHAR	prefix[PREFIXSIZE]	Prefix name	Required
ODM_CHAR	devid[DEVIDSIZE]	Device ID	Optional
ODM_SHORT	base	Base device flag	Required
ODM_SHORT	has_vpd	VPD flag	Required
ODM_SHORT	detectable	Detectable/non-detectable flag	Required
ODM_SHORT	chgstatus	Change status flag	Required
ODM_SHORT	bus_ext	Bus extender flag	Required
ODM_SHORT	inventory_only	Inventory only flag	Required
ODM_SHORT	fru	FRU flag	Required

Predefined Devices Object Class Fields			Part 2 of 2
ODM Type	**Descriptor Name**	**Description**	**Descriptor Status**
ODM_SHORT	led	LED value	Required
ODM_SHORT	setno	Set number	Required
ODM_SHORT	msgno	Message number	Required
ODM_VCHAR	catalog[CATSIZE]	Catalog file name	Required
ODM_CHAR	DvDr[DDNAMESIZE]	Device driver name	Optional
ODM_METHOD	Define	Define method	Required
ODM_METHOD	Configure	Configure method	Required
ODM_METHOD	Change	Change Method	Required
ODM_METHOD	Unconfigure	Unconfigure method	Optional*
ODM_METHOD	Undefine	Undefine method	Optional*
ODM_METHOD	Start	Start method	Optional
ODM_METHOD	Stop	Stop method	Optional
ODM_CHAR	uniquetype[UNIQUESIZE]	Unique type	Required

These fields have the following descriptions:

Device Type
The Device Type descriptor is derived from the product name or model number. For example, IBM 3812–2 Model 2 Page printer and IBM 4201 Proprinter II are two types of printer device types. Each device type supported by the system should have an entry in the PdDv object class.

Device Class
Associated functional class name. A Functional class is a group of device instances sharing the same high-level function. For example, **printer** is a functional class name representing all devices that generate hardcopy output.

Device Subclass
Identifies the device subclass associated with the device type. A device class can be partitioned into a set of device subclasses whose members share the same interface and typically are managed by the same device driver. For example, parallel and serial printers form two subclasses within the class of printer devices.

The configuration process uses the subclass to determine valid parent-child connections. For example, an rs232 8 port adapter has information that indicates that each of its eight ports only supports devices whose subclass is rs232. Also, the subclass for one device class can be a subclass for a different device class. In other words, several device classes can have the same device subclass.

Prefix Name
The Assigned Prefix in the Customized database, used to derive the device instance name and /**dev** name. For example, **tty** is a Prefix Name assigned to the tty port device type. Names of tty port instances would then look like `tty0`, `tty1`, or `tty2`. The rules for generating device instance names are given in the Customized Devices object class under the **Device Name** descriptor.

Device ID	Device ID describes card IDs for microchannel adapter cards. These card IDs are read from POS registers and uniquely identify the card type. The bus Configure method obtains the card IDs from the microchannel adapter cards and uses this descriptor to find the predefined information corresponding to the cards. The format is **0xAABB** where **AA** identifies the POS(0) value and **BB** the POS(1) value.
Base Device Flag	A base device is any device that forms part of a minimal base system. During the first phase of system boot, a minimal base system is configured to permit access to the root volume group and hence to the root file system.This minimal base system can include, for example, the standard I/O diskette adapter and a SCSI hard drive. This flag is not used to determine which devices are to be configured in the first phase of system boot. It serves only to identify at runtime which devices need to be updated in the boot image when configuration changes are made. A value of TRUE means that the device is a base device, and a value of FALSE that it is not.
VPD Flag	Certain devices contain Vital Product Data (VPD) that can be retrieved from the device itself. This attribute specifies whether device instances belonging to the device type contain extractable VPD or not. A value of TRUE means that the device has extractable VPD, and a value of FALSE that it does not.
Detectable/Nondetectable Flag	Specifies whether the device instance is detectable or nondetectable. A device whose presence and type can be electronically determined, once it is actually powered on and attached to the system, is said to be detectable. A value of TRUE means that the device is detectable, and a value of FALSE that it is not.
Change Status Flag	Indicates the initial value of the Change Status flag to be used in the Customized Devices (CuDv) object class. Refer to the corresponding descriptor in the CdDv object class for a complete description of this flag. A value of NEW means that the device is to be flagged as new, and a value of FALSE that it is to be flagged as *don't care.*
Bus Extender Flag	Indicates that the device is a bus extender. The Bus Configurator uses the Bus Extender Flag descriptor to determine whether it should directly invoke the device's Configure method. A value of TRUE means that the device is a bus extender, and a value of FALSE that it is not a bus extender. This flag is further described in Device Methods for Adapter Cards.

Inventory Only Flag	Distinguishes devices that are represented solely for their replacement algorithm from those that actually manage the system. There are several devices that are represented solely for inventory or diagnostic purposes. Racks, drawers, and planars represent such devices. A value of TRUE means that the device is used solely for inventory or diagnostic purposes, and a value of FALSE that it is not used solely for diagnostic or inventory purposes.
FRU Flag	Identifies the type of FRU (Field Replaceable Unit) for the device. The three possible values for this field are:

NO_FRU	Indicates that there is no FRU (for pseudo-devices).
SELF_FRU	Indicates that the device is its own FRU.
PARENT_FRU	Indicates that the FRU is the parent.

LED Value	Indicates the hex value to be displayed on the LEDs when the Configure method executes. Refer to *RISC System/6000 System Problem-Solving Guide* for a list of valid LED values.
Catalog File Name	Identifies the file name of the NLS message catalog that contains all messages pertaining to this device. This includes the device description and its attribute descriptions. All NLS messages are identified by a catalog file name, set number, and message number.
Set Number	Identifies the set number that contains all the messages for this device in the specified NLS message catalog. This includes the device description and its attribute descriptions.
Message Number	Identifies the message number in the specified set of the NLS message catalog. The message corresponding to the message number contains the textual description of the device.
Device Driver Name	Identifies the base name of the device driver associated with all device instances belonging to the device type. For example, a device driver name for a keyboard could be `ktsdd`. For the tape device driver, the name could be `tapedd`. The Device Driver Name can be passed as a parameter to the **loadext** routine to load the device driver, if the device driver is located in the **/etc/drivers** directory. If the driver is located in a different directory, the full path must be appended in front of the Device Driver Name before passing it as a parameter to the **loadext** subroutine.
Define Method	Name of the Define method associated with the device type. All Define method names start with the prefix **def**.
Configure Method	Name of the Configure method associated with the device type. All Configure method names start with the prefix **cfg**.
Change Method	Name of the Change method associated with the device type. All Change method names start with the prefix **chg**.

Unconfigure Method	Name of the Unconfigure method associated with the device type. All Unconfigure method names start with the prefix **ucfg**. **Note:** The Optional* descriptor status indicates that this field is optional for those devices (for example, the bus) that are never unconfigured or undefined. For all other devices, this descriptor is required.
Undefine Method	Name of the Undefine method associated with the device type. All Undefine method names start with the prefix **udef**. **Note:** The Optional* descriptor status indicates that this field is optional for those devices (for example, the bus) that are never unconfigured or undefined. For all other devices, this descriptor is required.
Start Method	Name of the Start method associated with the device type. All Start method names start with the prefix **stt**. The Start method is optional and only applies to devices that support the Stopped device state.
Stop Method	Name of the Stop method associated with the device type. All Stop method names start with the prefix **stp**. The Stop method is optional and only applies to devices that support the Stopped device state.
Unique Type	A key that is referenced by the **PdDvLn** link in CdDv object class. The key is a concatenation of the Device Class, Device Subclass, and Device Type values with a / (backslash) used as a separator. For example, for a class of `disk`, a subclass of `scsi`, and a type of `670mb`, the Unique Type is **disk/scsi/670mb**. This descriptor is needed so that a device instance's object in the CdDv object class can have a link to its corresponding PdDv object. Other object classes in both the Predefined and Customized databases also use the information contained in this descriptor.

Related Information

The **odmadd** command.

Customized Devices (CuDv) object class.

The **odm_add_obj** subroutine, **loadext** subroutine.

Writing a Define Method, Writing a Configure Method, Writing a Change Method, Writing an Undefine Method, Writing an Unconfigure Method, Writing Start and Stop Methods.

Object Data Manager (ODM) Overview, Understanding ODM Object Classes and Objects, Understanding ODM Descriptors in *General Programming Concepts.*

RISC System/6000 System Problem-Solving Guide.

Adapter-Specific Considerations for the PdDv Object Class

The information to be populated into the Predefined Devices Object Class is described in the Predefined Device (PdDv) object class. The following descriptors should be set as indicated:

Device Class	Set to `adapter`.
Device Subclass	Set to `mca`, which identifies it as an adapter card that can be plugged into the Microchannel bus.
Device ID	Must identify the values that are obtained from the POS(0) and POS(1) registers on the adapter card. The format is `0xAABB`, where `AA` is the hexadecimal value obtained from POS(0) and `BB` the value from POS(1). This descriptor is used by the Bus Configurator to match up the physical device with its corresponding information in the Configuration database.
Bus Extender Flag	Usually set to FALSE, which indicates that the adapter card is not a bus extender. This descriptor is set to TRUE for a multi-adapter card requiring different sets of bus resources assigned to each adapter. The Standard I/O Planar is an example of such a card.

The Bus Configurator behaves slightly differently for cards that are bus extenders. Typically, it finds an adapter card and returns the name of the adapter to the Configuration Manager so that it can be configured.

However, for a bus extender, the Bus Configurator directly invokes the device's Configure method. The bus extender's Configure method defines the various adapters on the card as separate devices (each needing its own Predefined information and device methods), and writes the names to standard output for the Bus Configurator to intercept. The Bus Configurator adds these names to the list of device names for which it is to assign bus resources.

An example of a type of adapter card that would be a bus extender is one which allows an expansion box with additional card slots to be connected to the system.

Related Information

Predefined Devices (PdDv) Object Class.

Adapter-Specific Considerations for the PdAt Object Class.

Writing a Configure Method.

Guide to Writing Device Methods

The following articles are provided as guidance for writing device methods.

Warning: These device method source code examples are only intended to assist in the development of a working software program. These examples do not function as written. Additional code is required.

General Information for Writing Device Methods

Returning Errors from Device Methods.

Loading a Device Driver.

Device Methods for Adapter Cards: Guidelines.

Adapter-Specific Considerations for the Predefined Devices (PdDv) Object Class.

Adapter-Specific Considerations for the Predefined Attributes (PdAt) Object Class.

Requirements for Individual Device Methods

Writing a Change Method.

Writing a Configure Method.

Writing a Define Method.

Writing a Unconfigure Method.

Writing a Undefine Method.

Writing Optional Stop and Start Methods.

How Device Methods Return Errors

Description

Device methods indicate errors to the Configuration Manager and runtime configuration commands by exiting with a nonzero exit code. The Configuration Manager and configuration commands can understand only the exit codes defined in the **cf.h** file.

Often more than one error code can describe any given error. This is because many exit codes correspond to highly specific errors, while others are more general. Whenever possible, use the most specific error code possible.

For example, if your Configure method obtains an attribute from the Customized Attributes (CuAt) object class for filling in the Device Dependent Structure (DDS) but the value is invalid (possibly due to a corrupted database), you might exit with the E_BADATTR error. On the other hand, you might choose the E_DDS exit code due to some other error condition while building the DDS.

Related Information

The Customized Attributes (CuAt) object class.

The Device Dependent Structure (DDS).

Writing a Device Method.

Adding an Unsupported Device to the System, Device Configuration Subsystem: Programming Introduction, Object Data Manager (ODM) Overview, Understanding Device Methods Interfaces in *Kernel Extensions and Device Support Programming Concepts.*

Loading a Device Driver

Description

The **loadext** subroutine is used to load and unload device drivers. The name of the device driver is passed as a parameter to the **loadext** routine. If the device driver is located in the **/etc/drivers** directory, just the device driver name without path information can be specified to the **loadext** subroutine. If the device driver is located in another directory, the fully qualified path name of the device driver must be specified.

The Device Driver Name descriptor of Predefined Devices (PdDv) object class objects is intended to contain only the device driver name and not the fully qualified path name. For device drivers located in the **/etc/drivers** directory, a Configure method can obtain the name of the driver from the Device Driver Name descriptor to pass to the **loadext** routine. This is convenient since most drivers are located in the **/etc/drivers** directory.

If a device driver is located in a directory other than **/etc/drivers**, the path name must be handled differently. The Configure method could be coded to assume a particular path name, or, for more flexibility, the path name could be stored as an attribute in the Predefined Attribute (PdAt) object class. The Configure method is responsible for knowing how to obtain the fully qualified path name to pass to the **loadext** subroutine.

Related Information

The **loadext** subroutine.

Writing a Configure Method.

Predefined Devices (PdDv) object class, Predefined Attribute (PdAt) object class.

Device Methods for Adapter Cards: Guidelines

Description

The device methods for an adapter card are essentially the same as for any other device. They need to perform roughly the same tasks as those described in Writing A Device Method. However, there is one additional important consideration. The Bus Configure method, or Bus Configurator, is responsible for discovering the adapter cards present in the system and for assigning bus resources to each of the adapters. These resources include interrupt levels, DMA arbitration levels, bus memory, and bus I/O space.

Adapters are typically defined and configured at boot time. However, if an adapter is not configured due to unresolvable bus resource conflicts, or if an adapter is unconfigured at run time with the **rmdev** command, an attempt to configure an adapter at run time may occur.

If an attempt is made, the Configure method for the adapter must take these steps to ensure system integrity:

1. Make sure that the card is actually present in the system by reading POS(0) and POS(1) from the slot that is supposed to contain the card and comparing the values with what they are supposed to be for the card.
2. Invoke the **busresolve** routine to ensure that the adapters bus resource attributes, as represented in the database, do not conflict with any of the configured adapters.

Additional guidelines must be followed when adding support for a new adapter card. They are discussed in:

- Adapter-Specific Considerations for the PdDv Object Class
- Adapter-Specific Considerations for the PdAt Object Class.

Related Information

The **rmdev** command.

Writing a Configure Method.

Adapter-Specific Considerations for the PdAt Object Class, Adapter-Specific Considerations for the PdDv Object Class.

Writing a Change Method

Syntax

chg*dev* **-l** *name* [**-p** *parent*] [**-w** *connection*] [**-P** | **-T**]
[**-a** *attr=value* [**-a** *attr=value*] ...]

Flags

-l *name*	Identifies the logical name of the device to be changed.
-p *parent*	Identifies the logical name of a new parent for the device. This flag is used to move a device from one parent to another.
-w *connection*	Identifies a new connection location for the device. This flag either identifies a new connection location on the device's existing parent, or if the **-p** option is also used, it identifies the connection location on the new parent device.
-P	Indicates that the changes are to be recorded in the Customized database without those changes being applied to the actual device. This is a useful flag for a device which is usually kept open by the system such that it can not be changed. Changes made to the database with this flag are later applied to the device when it is configured at system reboot.
-T	Indicates that the changes are to be applied only to the actual device and not recorded in the database. This is a useful flag for allowing temporary configuration changes that will not apply once the system is rebooted.
-a *attr=value*	Identifies an attribute to be changed and the value to which it should be changed.

Description

The Change method is responsible for applying configuration changes to a device. If the device is in the Defined state, the changes are simply recorded in the Customized database. If the device is in the Available state, the Change method must also apply the changes to the actual device by reconfiguring it.

Your Change method does not need to support all the flags described for Change methods. For instance, if your device is a pseudo-devices with no parent, it need not support parent and connection changes. Even for devices that have parents, it may be desirable to disallow parent and connection changes. For a printer, such changes may make sense since a printer is easily moved from one port to another. An adapter card, by contrast, is not usually moved without first shutting off the system. It is then automatically configured at its new location when the system is rebooted. Consequently, there may not be a need for a Change method to support parent and connection changes.

In deciding whether to support the **-T** and **-P** flags, remember that these flags will allow a device's configuration to get out of sync with the configuration database. The **-P** flag can often be useful for devices that are typically kept open by the system. The Change methods for most IBM-supported devices do not support the **-T** flag.

Writing a Change Method

In applying changes to a device in the Available state, your Change method could terminate the device from the driver, rebuild the device-dependent structure (DDS) using the new information, and define the device again to the driver using the new DDS. Your method may also need to reload adapter software or perform other device-specific operations. An alternative is to simply invoke the device's Unconfigure method, update the Customized database, and invoke the device's Configure method.

By convention, the first three characters of the name of the Change method should be **chg**. The remainder of the name can be any characters, subject to AIX file-name restrictions, that identify the device or group of devices which use the method.

Guidelines for Writing a Change Method

Note: This list of tasks is meant to serve as a guideline for writing a Change method. In writing a method for a specific device, you may be able to leave out some of the tasks. For instance, if your device does not support the changing of parent or connection, there is no need to include those tasks. You may also find that your device has special needs that are not listed in these tasks.

If your Change method is written to invoke the Unconfigure and Configure methods, it must:

1. Validate the input parameters. The **–l** flag must be supplied to identify the device that is to be undefined. You may want to exit with an error if options that your method does not support are specified.
2. Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the configuration database using the **odm_lock** subroutine. See Writing a Define Method for an example.
3. Retrieve the Customized Device (CuDv) object for the device to be changed by getting the CuDv object whose Device Name descriptor matches the name supplied with the **–l** option. If no object is found with the specified name, exit with an error.
4. Validate all attributes being changed. Make sure that the attributes apply to the specified device, that they can be set by the user, and that they are being set to valid values. The **attrval** subroutine can be used for this purpose. If you have attributes whose values depend on each other, you need to write the code to cross check them. If invalid attributes are found, your method needs to write information to standard error describing them. See Handling Invalid Attributes.
5. If a new parent device has been specified, find out whether it exists by querying the CuDv object class for an object whose Device Name descriptor matches the new parent name. If no match is found, exit with an error.
6. If a new connection has been specified, validate that this device can be connected there. Do this by querying the Predefined Connection (PdCn) object class for an object whose UniqueType descriptor matches the Link to the Predefined Devices Object Class descriptor of the parent's CuDv object, whose Connection Key descriptor matches the subclass name of the device being changed, and whose Connection Location descriptor matches the new connection value. If no match is found, exit with an error.

   If a match is found, the new connection is valid. If the device is currently available, then it should still be available after being moved to the new connection. Since only one device can be available at a particular connection, the Change method will need to check for other available devices already at that connection. If one is found, exit with an error.

7. If the device state is Available and the **–P** flag was not specified, invoke the device's Unconfigure method using the **odm_run_method** command. This fails if the device has available children, which is why the Change method does not need to check explicitly for children.
8. Record new attribute values in the database. If parent or connection changed, update the Parent Device Logical Name, Location Where Connected on Parent Device, and Location Code descriptors of the device's CuDv object.
9. If the device state was Available before being unconfigured, invoke the device's Configure method via the **odm_run_method** command. If this returns in error leaving the device unconfigured, you may want your Change method to restore the Customized database for the device to its pre-change state.
10. Ensure that all object classes are closed and terminate the ODM. Exit with an exit code of 0 (zero) if there were no errors.

Handling Invalid Attributes

If the Change method detects attributes that are in error, it must write information to the **stderr** file to identify them. This consists of writing the attribute name followed by the attribute description. Only one attribute and its description is to be written per line. If an attribute name was mistyped so that it does not match any of the device's attributes, write the attribute name supplied on a line by itself.

The **mkdev** and **chdev** configuration commands intercept the information written to standard error by the Change method. They in turn write it out following an error message describing that there were invalid attributes. Both the attribute name and attribute description are needed to identify the attribute. If you invoked the **mkdev** or **chdev** command directly, you can recognize the attributes by attribute name. If you are using SMIT, these comands recognize attributes by description.

The attribute description is obtained from the appropriate message catalog. A message is identified by catalog name, set number, and message number. The catalog name and set number are obtained from the device's Predefined Device (PdDv) object. The message number is obtained from the NLS Index descriptor in either the Predefined Attribute (PdAt) or Customized Attribute (CuAt) object corresponding to the attribute.

Related Information

The **chdev** command, **mkdev** command, **rmdev** command.

Customized Devices object class, Predefined Devices object class, Predefined Connection object class, Predefined Attribute object class, Customized Attribute object class.

Writing a Device Method.

The Unconfigure method Configure method .

The **attrval** subroutine, **odm_run_method** subroutine.

The Device Dependent Structure (DDS) Overview.

Object Data Manager (ODM) Overview in *General Programming Concepts.*

ODM Device Configuration Object Classes, Understanding Pseudo-Device Drivers, Understanding Device Dependencies and Child Devices in *Kernel Extensions and Device Support Programming Concepts.*

Writing a Configure Method

Syntax

cfg*dev* **–l** *name* [**–1** / **–2**]

Flags

–l *name*	Identifies the logical name of the device to be configured.
–1	Specifies that the device will be configured in phase 1 of system boot. This flag cannot be specified with the –2 option. If neither the –1 nor the –2 flags are specified, the Configure method is invoked at runtime.
–2	Specifies that the device will be configured in phase 2 of system boot. This flag cannot be specified with the –1 option. If neither the –1 nor the –2 flags are specified, the Configure method is invoked at runtime.

Description

The Configure method is responsible for configuring a device, that is, making it available for use in the system. It changes a device's state from Defined to Available. If the device has a device driver, the Configure method is responsible for loading the device driver into the kernel and describing the device characteristics to the driver. For an intermediate device (for example, a SCSI bus adapter), this method also determines which attached children are to be configured and writes their logical names to standard output.

The Configure method is invoked by either the **mkdev** configuration command or by the Configuration Manager. Because the Configuration Manager runs a second time in phase 2 system boot and can also be invoked repeatedly at runtime, a device's Configure method can be invoked to configure an already available device. This is not an error condition. In the case of an intermediate device, the Configure method should check again for the presence of child devices. If the device is not an intermediate device, the method simply returns.

By convention, the first three characters of the name of the Configure method should be **cfg**. The remainder of the name can be any characters, subject to AIX file-name restrictions, that identify the device or group of devices that use the method.

In general, the Configure method obtains all the information it needs about the device from the Configuration database. The options specifying the phase of system boot can be used to limit certain functions to specific phases.

If the device has a parent device, the parent must be configured first. The Configure method for a device should fail if its parent is not already in the Available state.

Guidelines for Writing a Configure Method

Note: This list of tasks is meant to serve as a guideline for writing a Configure method. In writing a method for a specific device, you may be able to leave out some of the tasks. For instance, if your device is not an intermediate device or does not have a device driver, your method can be written accordingly. You may also find that your device has special needs that are not listed in these tasks.

Your Configure method must:

1. Validate the input parameters. The **–l** logical name option must be supplied to identify the device that is to be configured. The **–1** and **–2** options cannot be supplied at the same time.
2. Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the Configuration database using the **odm_lock** subroutine. See Writing a Define Method for an example.
3. Retrieve the Customized Devices (CuDv) object for the device to be configured. This is done by getting the CuDv object whose Device Name descriptor matches the name supplied with the **–l** logical name option. If no object is found with the specified name, exit with an error.
4. Retrieve the Predefined Devices (PdDv) object for the device to be configured by getting the PdDv object whose Uniquetype descriptor matches the Link to Predefined Devices Object Class descriptor of the device's CuDv object.
5. If either the **–1** or **–2** option is specified, the Configure method should obtain the LED Value descriptor of the device's PdDv object and display the value on the system LEDs using the **setleds** subroutine. This specifies when the Configure method will execute at boot time. If the system hangs during configuration at boot time, the displayed LED value indicates which Configure method the hang occurred in.
6. If the device is already configured (that is, the Device State descriptor of the device's CuDv object indicates that the device is in the Available state), and is an intermediate device, the Configure method should skip to the task of detecting children devices. If the device is configured but is not an intermediate device, the Configure method should simply exit with no error.
7. If the device is still in the Defined state, the following tasks should be performed:
   a. If the device has a parent, the Configure method must ensure that the parent device exists and is in the available state. The method can look at the Parent Device Logical Name descriptor of the device's CuDv object to obtain the parent name. If the device does not have a parent, this descriptor should be a null string.

      Assuming that the device does have a parent, the Configure method should obtain the parent device's CuDv object and check the Device State descriptor. If the object does not exist or is not in the Available state, exit with an error.

      Another check must be made if the device has a parent device. The Configure method must make sure that no other device connected to the same parent at the same connection location has been configured. This case could arise, for example, when different printers are connected to the same port using a switch box. Each of the printers would have the same parent and connection, but only one could be configured at any given time.

      The Configure method can make this check by querying the CuDv object class for objects whose Device State descriptor is set to available and whose Parent Device Logical Name and Location Where Connected on Parent Device descriptors match those for the device being configured. If a match is found, exit with an error.

b. If the device is an adapter card and the Configure method has been invoked at run time (indicated by the absence of both the **–1** and **–2** options), the Configure method should ensure that the adapter card is actually present. This can be done by reading POS registers from the card. This is essential, because if the card is present, the Configure method must invoke the **busresolve** library routine to assign bus resources to the card and ensure that bus resources for the adapter do not conflict with other adapter cards in the system. If the card is not present or the **busresolve** routine fails to resolve bus resources, exit with an error.

   The POS registers are obtained by opening and accessing the **/dev/bus0** special file.

c. Determine whether or not the device has a device driver. The Configure method obtains the name of the device driver from the Device Driver Name descriptor of the device's PdDv object. If this descriptor is a null string, the device does not have a device driver.

d. If the device has a device driver, the Configure method will need to perform the following tasks:

   - First, load the device driver. The **loadext** subroutine can be used to do this. Loading a Device Driver has more information on loading the device driver.
   - Determine the device's major number using the **genmajor** subroutine. See Understanding Device Major and Minor Numbers for a Special File for more details.
   - Determine the device's minor number, possibly by using the **getminor** and **genminor** subroutines.
   - Create the device special files in the **/dev** directory if they do not already exist. Special files are created with the **mknod** subroutine.
   - Build the device-dependent structure (DDS) for the device. This structure contains the information that describes the device's characteristics to the device driver. The information is usually obtained from the device's attributes in the Configuration database. You may need to refer to the appropriate device driver information to determine what the device driver expects the DDS to look like. The Device Dependent Structure (DDS) Overview describes the DDS structure.
   - Use the **sysconfig** subroutine to initialize and pass the DDS to the device driver.
   - If there is code to be downloaded to the device, read in the required file and pass the code to the device through the interface provided by the device driver. The file to be downloaded might possibly be identified by a Predefined Attribute (PdAt) or Customized Attribute (CuAt) object. By convention, microcode files should be in the **/etc/microcode** directory while downloaded adapter software should be the **/etc/asw** directory.

e. After the tasks relating to the device driver are complete, or if the device did not have a device driver, the Configure method should determine if it needs to obtain vital product data (VPD) from the device. The VPD Flag descriptor of the device's PdDv object specifies whether or not it has VPD. See Handling Device Vital Product Data (VPD) for more details.

f. At this point, if no errors have been encountered, the device is configured. The Configure method should update the Device Status descriptor of the device's CuDv object to indicate that it is available.

8. If the device being configured is an intermediate device, the Configure method has one final task to perform. If the child devices actually attached can be detected, the Configure method is responsible for defining any new children not currently represented in the CuDv object class. This is accomplished by invoking the Define method for each new child device. For each detected child device that is already in the CuDv object class, the Configure method must look at the child device's CuDv Change Status Flag descriptor to see if it needs to be updated. If the descriptor's value is DONT_CARE, nothing needs to be done. If it has any other value, it must be set to SAME and the child device's CuDv object must be updated. The Change Status Flag descriptor is used by the system to indicate configuration changes.

   If the device is an intermediate device but cannot detect attached children, it can query the CuDv object class for children. The value of the Change Status Flag descriptor for these child devices should be DONT_CARE since the parent device cannot detect them. Sometimes a child device has an attribute specifying to the Configure method whether the child is to be configured. The **autoconfig** attribute of TTY devices is an example of this type of attribute.

   Regardless of whether the child devices are detectable, the Configure method should write the device logical names of the children to be configured to standard output, separated by space characters. If the method was invoked by the Configuration Manager, the Manager invokes the Configure method for each of the child device name written to standard output.

9. Finally, ensure that all object classes are closed and terminate the ODM. Exit with an exit code of 0 (zero) if there are no errors.

Handling Device Vital Product Data (VPD)

Devices that provide vital product data (VPD) should be identified in the Predefined Device (PdDv) object class by setting the VPD Flag descriptor to TRUE in each of the device's PdDv objects. The Configure method must obtain the VPD from the device and store it into the Customized VPD (CuVPD) object class. The appropriate hardware documentation for the device should be consulted to determine how to retrieve the device's VPD. In many cases, VPD can be obtained for a device from the device driver with the **sysconfig** subroutine.

Once the VPD is obtained from the device, the Configure method should query the CuVPD object class to see if the device already has hardware VPD stored there. If there is, the method should compare the VPD obtained from the device with that from the CuVPD object class. If the VPD is the same in both cases, no further processing is needed. If they are different, update the VPD in the CuVPD object class for the device. If there is no VPD in the CuVPD object class for the device, add the device's own VPD into it.

Comparing the device's VPD with that in the CuVPD object class first helps make modifications to the CuVPD object class less frequent. This results from the fact that the VPD from a device typically does not change. Reducing the number of database writes increases performance and minimizes the possibility of data loss.

Understanding Configure Method Errors

For many of the errors detected by the Configure method, the method can simply exit with the appropriate exit code. In other cases, the Configure method may need to undo some operations it has performed. For instance, after loading the device's device driver and defining the device to the device driver by passing it the Device Dependent Structure (DDS), the Configure method may subsequently encounter an error while downloading microcode. If this happens, the method should terminate the device from the device driver with the **sysconfig** subroutine and unload the driver with the **loadext** subroutine.

The Configure method does not need to delete the special files or unassign the major and minor numbers if the major and minor numbers were successfully allocated and the special file created before the error was encountered.

This is because the AIX configuration scheme allows both major and minor numbers and special files to be maintained for a device even though the device is unconfigured. If the device is configured again, the Configure method should recognize that the major and minor numbers are already allocated and that the special files already exist. See Understanding Major and Minor Numbers for a Special File for more information.

By the time the Configure method checks for child devices, it has already successfully configured the device that it was called to configure. Errors that occur while checking for child devices are indicated with the E_FINDCHILD exit code. The **mkdev** command detects whether the Configure method completed successfully. It can still display a message indicating that an error occurred while looking for child devices.

Related Information

The **mkdev** command.

The /**dev/bus0** special file.

Understanding Configure Method Errors, Writing an Unconfigure Method, Writing a Define Method.

Customized Devices (CuDv) object class, Predefined Devices (PdDv) object class, Customized Attributes (CuAt) object class, Predefined Attribute (PdAt) object class, Customized Vital Product Data (CuVPD) object class.

The **loadext** subroutine, **genmajor** subroutine, **getminor** subroutine, **genminor** subroutine, **sysconfig** subroutine, **odm_initialize** subroutine, **odm_lock** subroutine, **mknod** subroutine, **reldevno** subroutine, **relmajor** subroutine.

Device-Dependent Structure (DDS) Overview.

Handling Device Vital Product Data (VPD).

Writing a Device Method.

Special File Overview in *Files Reference.*

Object Data Manager (ODM) Overview in *General Programming Concepts.*

Understanding Device States, Understanding Major and Minor Numbers For A Special File, Understanding Device Dependencies and Child Devices, Loading A Device Driver Configuration Manager Overview, Understanding System Boot Processing, Device Driver Kernel Extension Overview in *Kernel Extensions and Device Support Programming Concepts.*

The General Information section of *RISC System/6000 Hardware Technical Reference* provides more details on the VPD.

Writing a Define Method

Syntax

def*dev* **–c** *class* **–s** *subclass* **–t** *type* [**–p** *parent* **–w** *connection*] [**–l** *name*]

Flags

–c *class*	Specifies the class of the device being defined. Class, subclass, and type are required to identify the Predefined Device object in the Predefined Device (PdDv) object class for which a customized device instance is to be created.
–s *subclass*	Specifies the subclass of the device being defined. Class, subclass, and type are required to identify the Predefined Device object in the PdDv object class for which a customized device instance is to be created.
–t *type*	Specifies the type of the device being defined. Class, subclass, and type are required to identify the predefined device object in the PdDv object class for which a customized device instance is to be created.
–p *parent*	Specifies the logical name of the parent device. This logical name is required for devices that connect to a parent device. This flag does not apply to devices that do not have parents; for example, most pseudo-devices.
–w *connection*	Specifies where the device connects to the parent. This flag applies only to devices that connect to a parent device.
–l *name*	This flag is passed in by the **mkdev** command if the user invoking the command is defining a new device and wants to select the name for the device. The Define method assigns this name as the logical name of the device in the Customized Devices (CuDv) object, if the name is not already in use. If this option is not specified, the Define method generates a name for the device. Not all devices support or need to support this flag.

Description

The Define method is responsible for creating a customized device instance of a device in the Customized database. It does this by adding an object for the device into the Customized Devices (CuDv) object class. The Define method is invoked either by the **mkdev** configuration command, by a node configuration program, or by the Configure method of a device that is detecting and defining child devices.

By convention, the first three characters of the name of the Define method should be **def**. The remainder of the name can be any characters that identify the device or group of devices that use the method, subject to AIX file name restrictions.

The Define method uses information supplied as input, as well as information in the Predefined database, for filling in the CuDv object. If the method is written to support a single device, it can ignore the class, subclass, and type options. In contrast, if the method supports multiple devices, it may need to use these options to obtain the PdDv device object for the type of device being customized.

Guidelines for Writing a Define Method

Note: This list of tasks is meant to serve as a guideline for writing a Define method. In writing a method for a specific device, you may be able to leave out some of the tasks. For instance, if your device does not have a parent, there is no need to include all of the parent and connection validation tasks. You may also find that your device has special needs that are not listed in these tasks.

Your Define method must:

1. Validate input parameters.

   Generally, a Configure method that invokes a Define method to define a child device is coded to pass the options expected by the child device's Define method. However, the **mkdev** command always passes the class, subclass, and type options, while only passing the other options based on user input to the **mkdev** command. Thus, the Define method may need to ensure that all of the options it requires have been supplied to it. For example, if the Define method expects parent and connection options for the device being defined, it should ensure that the options are indeed supplied. Also, a Define method that does not support the **–l** name specification option may want to exit with an error if the option is supplied.

2. Initialize the Object Data Manager (ODM).

   Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the configuration database using the **odm_lock** subroutine. The following code fragment illustrates this process:

   ```
   #include <cf.h>

   if (odm_initialize() < 0)
           exit(E_ODMINIT);        /* initialization failed */

   if (odm_lock("/etc/objrepos/config_lock",0) == -1) {
           odm_terminate();
           exit(E_ODMLOCK);        /* database lock failed */
   }
   ```

3. Retrieve the predefined PdDv object for the type of device being defined.

   This is done by obtaining the object from the PdDv object class whose Class, Subclass, and Type descriptors match the class, subclass, and type options supplied to the Define method. If no match is found, the Define method should exit with an error. Information will be taken from the PdDv device object in order to create the CuDv device object.

4. Ensure that the parent device exists.

   If the device being defined connects to a parent device and the name of the parent has been supplied, the Define method must ensure that the specified device actually exists. It does this by retrieving the CuDv object whose Device Name descriptor matches the name of the parent device supplied using the **–p** flag. If no match is found, the Define method should exit with an error.

5. Validate that the device being defined can be connected to the specified parent device.

   If the device has a parent and that parent device exists in the CuDv object class, you must next validate that the device being defined can be connected to the specified parent device. To do this, retrieve the predefined connection object from the Predefined Connection (PdCn) object class whose UniqueType, Connection Key, and Connection Location descriptors match the Link To Predefined Devices Object Class descriptor of the parent's CuDv object obtained in the previous step and the subclass and connection options input into the Define method, respectively. If no match is found, an invalid connection has been specified. This may be because the specified parent is not an intermediate device, does not accept the type of device being defined (as described by subclass), or does not have the connection location identified by the connection option.

6. Assign a logical name to the device.

   Each newly assigned logical name must be unique to the system. If a name has been supplied using the **–l** flag, you must make sure it is unique before assigning it to the device. This is done by checking the CuDv object class for any object whose Device Name descriptor matches the desired name. If a match is found, the name is already used and the Define method must exit with an error.

   If the Define method is to generate a name, it can do so by obtaining the prefix name from the Prefix Name descriptor of the device's PdDv device object and invoking the **genseq** subroutine to obtain a unique sequence number for this prefix. By appending the sequence number to the prefix name, a unique name results. The **genseq** routine looks in the CuDv object class to ensure that it assigns a sequence number that has not been used with the specified prefix to form a device name.

   In some cases, a Define method may need to ensure that only one device of a particular type has been defined. For example, there can only be one PTY device customized in the CuDv object class. The PTY Define method does this by querying the CuDv object class to see if a device by the name `pty0` exists. If it does, the PTY device has already been defined. Otherwise, the Define method proceeds to define the PTY device using the name `pty0`.

7. Determine the device's location code.

   If the device being defined is a physical device, it has a location code. Understanding Location Codes has more information about location codes.

8. Create the new CuDv object.

   Set the CuDv descriptors as follows:

device name	Use the name as determined above.
device status flag	Set to the Defined state.
change status flag	Set to the same value as that found in the Change Status Flag descriptor in the device's PdDv object.
device driver instance	Typically set to the same value as the Device Driver Name descriptor in the device's PdDv object. It may be used later by the Configure method.
device location code	Set to a null string if the device does not have a location code. Otherwise, set it to the value computed.

parent device logical name
: Set to a null string if the device does not have a parent. Otherwise set it to the parent name as specified by the parent option.

location where connected on parent device
: Set to a null string if the device does not have a parent. Otherwise, set it to the value specified by the connection option.

link to predefined devices object class
: Set to the value obtained from the Unique Type descriptor of the device's PdDv object.

9. Write the name of the device to standard output.

   A blank should be appended to the device name to serve as a separator in case other methods write device names to standard output. Either the **mkdev** command or the Configure method that invoked the Define method will intercept standard output to obtain the device name assigned to the device.

10. Ensure all object classes are closed and terminate the ODM.

    Exit with an exit code of 0 (zero) if there were no errors.

Related Information

The **mkdev** command.

Writing an Undefine Method, Writing a Configure Method, Loading A Device Driver.

Customized Devices (CuDv) object class, Predefined Devices (PdDv) object class, Predefined Connection (PdCn) object class.

The **genseq** subroutine, **odm_initialize** subroutine, **odm_lock** subroutine.

Understanding Device States, Understanding Device Classes, Subclasses, and Types, Understanding Major and Minor Numbers for a Special File, Understanding Device Dependencies and Child Devices, Understanding Pseudo-Device Drivers, Configuration Manager Overview, Understanding System Boot Processing, Device Driver Kernel Extension Overview, Writing a Device Method Overview in *Kernel Extensions and Device Support Programming Concepts.*

Devices Overview for System Management, Object Data Manager (ODM) Overview, Understanding Location Codes in *General Concepts and Procedures*

Writing an Unconfigure Method

Syntax

ucfg*dev* **-l** *name*

Flags

-l *name*	Identifies the logical name of the device to be unconfigured.

Description

The Unconfigure method is responsible for unconfiguring an available device. This means taking a device that is available for use by the system and making it unusable. All the customized information about the device is to be retained in the database so that the device can be configured again exactly as it was before.

The actual operations required to make a device no longer available for use depend on what the Configure Method did to make the device available in the first place. For instance, if the device has a device driver, the Configure method will have Loading a Device Driver into the kernel and described the device to the driver through a Device Dependent Structure (DDS).. The Unconfigure method thus needs to tell the driver to delete the device instance and then request an unload of the driver.

If the device is an intermediate device, the Unconfigure method must check the states of the child devices. If any child is in the Available state, the Unconfigure method will fail and leave the device configured. To ensure proper system operation, all children must be unconfigured before the parent can be unconfigured.

Although the Unconfigure method must check child devices, it does not need to check for device dependencies recorded in the Customized Dependency (CuDep) object class. See Understanding Device Dependencies and Child Devices.

The Unconfigure method must also fail if the device is currently open. In this case, the device driver must return a value for the **errno** variable of EBUSY to the Unconfigure method when the method requests the driver to delete the device. The device driver is the only component at that instant that knows the device is open. As in the case of configured children, the Unconfigure method will fail and leave the device configured.

When requesting the device driver to terminate the device, **errno** values other than EBUSY can be returned. The driver should return ENODEV if it does not know about the device. Under the best circumstances, however, this case should not occur. If ENODEV is returned, the Unconfigure method should go ahead and unconfigure the device with respect to the database so that the database and device driver are in agreement. If the device driver chooses to return any other **errno** value, it must still delete any stored characteristics for the specified device instance. The Unconfigure method should also indicate that the device is unconfigured by setting the state to Defined.

The Unconfigure method does not generally release the major number and minor number assignments for a device, nor does it delete the device's special files in the /**dev** directory. Understanding Major and Minor Numbers has more information on device methods, major numbers, minor numbers, and special files.

By convention, the first four characters of the name of the Unconfigure method should be **ucfg**. The remainder of the name can be any characters, subject to AIX file-name restrictions, that identify the device or group of devices that use the method.

Guidelines for Writing an Unconfigure Method

Note: This list of tasks is meant to serve as a guideline for writing an Unconfigure method. In writing a method for a specific device, you may be able to leave out some of the tasks. For instance, if your device is not an intermediate device or does not have a device driver, your method can be written accordingly. You may also find that your device has special needs that are not listed in these tasks.

Your Unconfigure method must:

1. Validate the input parameters. The **–l** flag must be supplied to identify the device that is to be unconfigured.
2. Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the Configuration database using the **odm_lock** subroutine. See Writing a Define Method for an example.
3. Retrieve the Customized Device (CuDv) object for the device to be unconfigured. This is done by getting the CuDv object whose Device Name descriptor matches the name supplied with the **–l** flag. If no object is found with the specified name, exit with an error.
4. Check the device's current state. If the Device Status descriptor indicates that the device is in the Defined state, then it is already unconfigured. You should exit as for a successful completion.
5. Check for child devices in the Available state. This can be done by querying the CuDv object class for objects whose Parent Device Logical Name descriptor matches this device's name and whose Device Status descriptor is not `defined`. If a match is found, exit with an error.
6. Retrieve the predefined Predefined Device (PdDv) object for the device to be configured by getting the PdDv object whose UniqueType descriptor matches the Link to Predefined Devices Object Class descriptor of the device's CuDv object. This object will be used to get the device driver name.
7. Determine whether the device has a device driver. The Unconfigure method obtains the name of the device driver from the Device Driver Name descriptor of the device's PdDv object. If this descriptor is a null string, the device does not have a device driver. In this case, skip to the task of updating the device's state.
8. If the device has a device driver, the Unconfigure method will need to perform the following tasks:
   a. Determine the device's major and minor numbers using the **genmajor** and **getminor** subroutines. These are used to compute the device's devno, using the **makedev** macro defined in the **sysmacros.h** header file, in preparation for the next task. See Understanding Device Major and Minor Numbers for a Special File for more details.
   b. Use the **sysconfig** subroutine to tell the device driver to terminate the device. If a value of EBUSY for the **errno** variable is returned, exit with an error.
   c. Use the **loadext** routine to unload the device driver from the kernel. The **loadext** routine will not actually unload the driver if there is another device still configured for the driver. See Loading a Device Driver for more details.

9. The device is now unconfigured. The Unconfigure method should update the Device Status descriptor of the device's CuDv object to `defined`.
10. Ensure that all object classes are closed and terminate the ODM. If there are no errors, exit with an exit code of 0 (zero).

Related Information

The **mkdev** command.

Writing a Configure Method, Writing a Device Method.

Customized Devices (CuDv) object class, Predefined Devices (PdDv) object class.

The **loadext** subroutine, **genmajor** subroutine, **getminor** subroutine, **sysconfig** subroutine, **odm_initialize** subroutine, **odm_lock** subroutine.

The Device Dependent Structure (DDS) Overview.

Special File Overview in *Files Reference*.

Object Data Manager (ODM) Overview in *General Programming Concepts*.

Understanding Device States, Understanding Major and Minor Numbers for a Special File, Understanding Device Dependencies and Child Devices, Loading a Device Driver in *Kernel Extensions and Device Support Programming Concepts*

Writing an Undefine Method

Syntax

udef*dev* **–l** *name*

Flags

–l *name*	Identifies the logical name of the device to be undefined.

Description

The Undefine method is responsible for deleting a Defined device from the Customized database. Once a device is deleted, it cannot be configured until it is once again defined by the Define method.

The Undefine method is also responsible for releasing the major and minor number assignments assignments for the device instance and deleting the device's special files from the /**dev** directory. If minor-number assignments are registered with the **genminor** subroutine, the Undefine method can release the major and minor number assignments and delete the special files by simply calling the **reldevno** subroutine.

By convention, the first four characters of the name of the Undefine method are to be **udef**. The remainder of the name can be any characters, subject to AIX file-name restrictions, that identify the device or group of devices that use the method.

Guidelines foe Writing an Undefine Method

Note: This list of tasks is meant to serve as a guideline for writing an Undefine method. You may find that your device has special needs that are not listed in these tasks.

Your Undefine method must:

1. Validate the input parameters. The **–l** flag must be supplied to identify the device that is to be undefined.
2. Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the configuration database using the **odm_lock** subroutine. See Writing a Device Method for an example.
3. Retrieve the Customized Device (CuDv) object for the device to be unconfigured. This is done by getting the CuDv object whose Device Name descriptor matches the name supplied with the **–l** flag. If no object is found with the specified name, exit with an error.
4. Check the device's current state. If the Device Status descriptor indicates that the device is not in the Defined state, then it is not ready to be undefined. If this is the case, exit with an error.
5. Check for any child devices. This check is accomplished by querying the CuDv object class for any objects whose Parent Device Logical Name descriptor matches this device's name. If the device has any children at all, regardless of the states they are in, the Undefine method must fail. All children must be undefined before the parent can be undefined.

6. Check to see if this device is listed as a dependency of another device. This is done by querying the Customized Dependency (CuDep) object class for objects whose Dependency descriptor matches this device's logical name. If a match is found, exit with an error. A device may not be undefined if it has been listed as a dependency by another device. Understanding Device Dependencies and Child Devices discusses dependencies.
7. If no errors have been encountered, the method can begin deleting customized information. First, delete the special files from the /**dev** directory. Next, delete all minor number assignments. If the last minor number has been deleted for a particular major number, release the major number as well, using the **relmajor** subroutine. The Undefine method should never delete objects from the Customized Device Driver (CuDvDr) object class directly, but should always use the routines provided. If the minor-number assignments are registered with the **genminor** subroutine, all of the above can be accomplished by the **reldevno** subroutine.
8. Delete all attributes for the device from the Customized Attribute (CuAt) object class. Simply delete all CuAt objects whose Device Name descriptor matches this device's logical name. It is not an error if the ODM routines used to delete the attributes indicate that no objects were deleted. This simply indicates that the device has no attributes that had been changed from the default values.
9. Delete the Customized Device (CuDv) object for the device.
10. Make sure all object classes are closed and terminate the ODM. Exit with an exit code of 0 (zero) if there are no errors.

Related Information

Writing a Define Method.

Customized Device (CuDv) object class, Customized Attributes (CuAt) object class, Customized Dependency (CuDep) object class, Customized Device Driver (CuDvDr) object class.

The **genminor** subroutine, **reldevno** subroutine, **relmajor** subroutine, **odm_initialize** subroutine, **odm_lock** subroutine.

Writing Optional Start and Stop Methods

Syntax

stt*dev* **–l** *name*
stp*dev* **–l** *name*

Flags

–l *name*	Identifies the logical name of the device to be started or stopped.

Description

The Start and Stop methods are optional methods. They allow a device to support the additional device state of Stopped. The Start method takes the device from the Stopped state to the Available state. The Stop method takes the device from the Available state to the Stopped state. Most devices do not have Start and Stop methods.

The Stopped state provides a state in which the device is configured in the system but unusable by applications. In this state, the device's driver is loaded and the device is defined to the driver. This might be implemented by having the Stop method issue a command telling the device driver not to accept any normal I/O requests. If an application subsequently issues a normal I/O request to the device, it will fail. The Start method can then issue a command to the driver telling it to start accepting I/O requests once again.

If you write Start and Stop methods for your device, your other methods must be written to account for the Stopped state. For instance, if one of your methods checks for a device state of Available, it might now need to check for both Available and Stopped states.

Additionally, write your Configure method so that it takes the device from the Defined state to the Stopped state. However, you can have the Configure method invoke the Start method, thus taking the device to the Available state. The Unconfigure method should be able to take the device to the Defined state from either the Available or Stopped states.

By convention, the first three characters of the name of the Start method should be **stt**. The first three characters of the name of the Stop method should be **stp**. The remainder of the names can be any characters, subject to AIX file-name restrictions, that identify the device or group of devices that use the methods.

Start and Stop methods, when they are used, are usually highly device-specific.

Related Information

Writing an Unconfigure Method, Writing a Configure Method.

Chapter 5. Communications Subsystem

ddclose Communications PDH Entry Point

Purpose

Frees up system resources used by the specified communications device until they are needed again.

Syntax

#include <sys/device.h>

int ddclose (*devno*, *chan*)
dev_t *devno*;
int *chan*;

Parameters

devno	Major and minor device numbers.
chan	Channel number assigned by the device handler's **ddmpx** entry point.

Description

The **ddclose** entry point frees up system resources used by the specified device until they are needed again. Data retained in the receive queue, transmit queue, or status queue is purged. All buffers associated with this channel are freed. The **ddclose** entry point should be called once for each successfully issued **ddopen** entry point.

Before issuing a **ddclose** entry point, a CIO_HALT operation should be issued for each previously successful CIO_START operation on this channel.

Execution Environment

A **ddclose** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return code for the **ddclose** entry point is the following:

ENXIO	Indicates an attempt to close an unconfigured device.

Related Information

The **ddmpx** entry point, **ddopen** entry point.

The CIO_HALT operation, CIO_START operation.

ddioctl (CIO_GET_STAT) Communications PDH Entry Point

Purpose

Returns the next status block in the status queue to a user-mode process.

Syntax

#include <sys/device.h>
#include <sys/comio.h>

int ddioctl (*devno*, *op*, *parmptr*, *devflag*, *chan*, *ext*)
dev_t *devno*;
int *op*;
struct status_block **parmptr*;
ulong *devflag*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
op	Indicates CIO_GET_STAT.
parmptr	Points to a **status_block** structure.
devflag	Set to the **DKERNEL** flag. This flag must be clear, indicating a call by a user-mode process.
chan	Specifies the channel number assigned by the device handler's **ddmpx** entry point.
ext	Device-dependent.

Description

Note: This operation should not be called by kernel-mode processes.

The CIO_GET_STAT operation returns the next status block in the status queue to a user-mode process.

Execution Environment

A CIO_GET_STAT operation can be called from the process environment only.

Return Values

In general, communication device handlers use the common codes defined for an operation. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the CIO_GET_STAT operation are the following:

ENXIO	Indicates an attempt to use an unconfigured device.
EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates an invalid parameter.
EACCES	Indicates an invalid call from a kernel process.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENODEV	Indicates that the device does not exist.

Related Information

The **ddioctl** entry point, **ddmpx** entry point.

Status Blocks for Communications Device Handlers.

Status Blocks for Communications Device Handlers

Status blocks are used to communicate status and exception information.

User-mode processes receive a status block whenever they request a CIO_GET_STAT operation. A user-mode process can wait for the next available status block by issuing a **ddselect** entry point with the specified POLLPRI event.

A kernel-mode process receives a status block through the **stat_fn** procedure. This procedure is specified when the device is opened with the **ddopen** entry point.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, CIO_START_DONE). A status block's options depend on the block code. The C structure of a status block is defined in the **<sys/comio.h>** file.

The following are the six common status codes:

- CIO_START_DONE
- CIO_HALT_DONE
- CIO_TX_DONE
- CIO_NULL_BLK
- CIO_LOST_STATUS
- CIO_ASYNC_STATUS.

Additional device-dependent status block codes may be defined.

CIO_START_DONE Status Block

This block is provided by the device handler when the CIO_START operation completes:

option[0]	The CIO_OK or CIO_HARD_FAIL status/exception code from the common or device-dependent list.
option[1]	The low-order two bytes are filled in with the **netid** field. This field is passed when the CIO_START operation is invoked.
option[2]	Device-dependent.
option[3]	Device-dependent.

CIO_HALT_DONE Status Block

This block is provided by the device handler when the CIO_HALT operation completes:

option[0]	The CIO_OK status/exception code from the common or device-dependent list.
option[1]	The low-order two bytes are filled in with the **netid** field. This field is passed when the CIO_START operation is invoked.
option[2]	Device-dependent.
option[3]	Device-dependent.

CIO_TX_DONE Status Block

The following block is provided when the physical device handler (PDH) is finished with a transmit request for which acknowledgment was requested:

option[0]	The CIO_OK or CIO_TIMEOUT status/exception code from the common or device-dependent list.
option[1]	The **write_id** field specified in the **write_extension** structure passed in the *ext* parameter to the **ddwrite** entry point.
option[2]	For a kernel-mode process, indicates the mbuf pointer for the transmitted frame.
option[3]	Device-dependent.

CIO_NULL_BLK Status Block

This block is returned whenever a status block is requested but there are none available:

option[0]	Not used
option[1]	Not used
option[2]	Not used
option[3]	Not used.

CIO_LOST_STATUS Status Block

This block is returned once after one or more status blocks is lost due to status queue overflow. The CIO_LOST_STATUS block provides the following:

option[0]	Not used
option[1]	Not used
option[2]	Not used
option[3]	Not used.

CIO_ASYNC_STATUS Status Block

This status block is used to return status and exception codes that occur unexpectedly:

option[0]	The CIO_HARD_FAIL or CIO_LOST_DATA status/exception code from the common or device-dependent list.
option[1]	Device-dependent
option[2]	Device-dependent
option[3]	Device-dependent.

Related Information

The **ddwrite** entry point, **ddselect** entry point.

The CIO_HALT operation, CIO_GET_STAT operation, CIO_START operation.

List of Common Status/Exception Codes.

ddioctl (CIO_HALT) Communications PDH Entry Point

Purpose

Removes the caller's network ID and undoes whatever was effected by the corresponding CIO_START operation.

Syntax

#include <sys/device.h>
#include <sys/comio.h>

int ddioctl (*devno*, *op*, *parmptr*, *devflag*, *chan*, *ext*)
dev_t *devno*;
int *op*;
struct session_blk **parmptr*;
ulong *devflag*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
op	Equals CIO_HALT.
parmptr	Points to a **session_blk** structure. This structure is defined in the **<sys/comio.h>** header file.
devflag	Set by kernel-mode processes to the **DKERNEL** flag. This flag is cleared by calling user-mode processes.
chan	Specifies the channel number assigned by the device handler's **ddmpx** routine.
ext	Device-dependent.

Description

The CIO_HALT operation must be supported by each physical device handler in the communication I/O subsystem. This entry point should be issued once for each successfully issued CIO_START operation. This entry point removes the caller's network ID and undoes whatever was effected by the corresponding CIO_START operation.

The CIO_HALT call returns immediately to the caller, before the operation completes. If the return indicates no error, the PDH builds a CIO_HALT_DONE status block upon completion. For kernel-mode processes, the status block is passed to the associated status function (specified at open time). For user-mode processes, the block is placed in the associated status/exception queue.

The session_blk Parameter Block

For the CIO_HALT operation, the *ext* parameter may be a pointer to a **session_blk** structure. This structure is defined in the **<sys/comio.h>** header file and the contains following fields:

status Indicates the status of the port. This field may contain additional information about the completion of the CIO_HALT operation. Besides the status codes listed here, device-dependent codes can be returned:

CIO_OK	Indicates that the operation was successful.

CIO_INV_CMD	Indicates that an invalid command was issued.
CIO_NETID_INV	Indicates that the network ID was invalid.

The **status** field is used for specifying immediately detectable erros. If the status is CIO_OK, the CIO_HALT_DONE status block should be processed to detemine whether the halt completed without errors.

netid Contains the network ID to halt.

Execution Environment

A CIO_HALT operation can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an operation. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the CIO_HALT operation are the following:

ENXIO	Indicates an attempt to use an unconfigured device.
EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates an invalid parameter.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENODEV	Indicates that the device does not exist.

Related Information

The **ddioctl** entry point.

The CIO_START operation, CIO_GET_STAT operation.

ddioctl (CIO_START) Communications PDH Entry Point

Purpose

Syntax

#include <sys/device.h>
#include <sys/comio.h>

int ddioctl (*devno*, *op*, *parmptr*, *devflag*, *chan*, *ext*)
dev_t *devno*;
int *op*;
struct session_blk **parmptr*;
ulong *devflag*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
op	Set to CIO_START.
parmptr	Points to **session_blk** structure.
devflag	Set to the **DKERNEL** flag by calling kernel-mode processes. This flag is cleared by calling user-mode processes.
chan	Specifies the channel number assigned by the device handler's **ddmpx** entry point.
ext	Device-dependent.

Description

The CIO_START communications PDH entry point must be supported by each physical device handler in the communication I/O subsystem. Its use varies from adapter to adapter. This entry point opens a communication session on a channel that has been opened by a **ddopen** entry point. Once a channel is opened, multiple CIO_START operations can be issued. For each successful start, a corresponding CIO_HALT operation must be issued later.

The CIO_START entry point requires only the *netid* input parameter. This parameter is registered for the session. At least one network ID must be registered for this session before the PDH successfully accepts a call to the **ddwrite** or **ddread** entry point on this session. If this start is the first issued for this port or adapter, the appropriate hardware initialization is performed. Time consuming initialization activities, such as call connection, are also performed.

This call returns immediately to the caller before the asynchronous command completes. If the return indicates no error, the PDH builds a CIO_START_DONE status block upon completion. For kernel-mode processes, the status block is passed to the associated status function (specified at open time). For user-mode processes, the status block is placed in the associated status/exception queue.

The session_blk Parameter Block

For the CIO_START operation, the *ext* parameter may be a pointer to a **session_blk** structure. This structure is defined in the **<sys/comio.h>** file and contains the following fields:

status	Indicates the status of the port. This field may contain additional information about the completion of the CIO_START operation. Besides the status codes listed here, device-dependent codes can be returned:

CIO_OK	Indicates that the operation was successful.
CIO_INV_CMD	Indicates that an invalid command was issued.
CIO_NETID_INV	Indicates that the network ID was invalid.
CIO_NETID_DUP	Indicates that the network ID was a duplicate of an existing ID already in use on the network.
CIO_NETID_FULL	Indicates that the network table is full.

netid	Contains the network ID to register with the start.

Execution Environment

A CIO_START operation can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the CIO_START operation are the following:

ENXIO	Indicates an attempt to use an unconfigured device.
EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates an invalid parameter.
ENOSPC	Indicates that the network ID table is full.
EADDRINUSE	Indicates a duplicate network ID.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENODEV	Indicates that the device does not exist.

Related Information

The **ddioctl** entry point, **ddwrite** entry point, **ddread** entry point.

The CIO_HALT operation, CIO_GET_STAT operation.

ddioctl (CIO_QUERY) Communications PDH Entry Point

Purpose

Returns various statistics from the device.

Syntax

#include <sys/device.h>
#include <sys/comio.h>

int ddioctl (*devno*, *op*, *parmptr*, *devflag*, *chan*, *ext***)**
dev_t *devno*;
int *op*;
struct query_parms **parmptr*;
ulong *devflag*;
int *chan*, *ext*;

Parameters

devno	Major and minor device numbers.
op	Set to CIO_QUERY.
parmptr	Points to a **query_parms** structure. This structure is defined in the **<sys/comio.h>** header file.
devflag	Set to the **DKERNEL** flag by calling kernel-mode processes. This flag is cleared by calling user-mode processes.
chan	Specifies channel number assigned by the device handler's **ddmpx** entry point.
ext	Device–dependent.

Description

The CIO_QUERY operation returns various statistics from the device. Counters are zeroed by the physical device handler when the device is configured. The data returned consists of two contiguous portions. The first portion contains counters to be collected and maintained by all device handlers in the communication I/O subsystem. The second portion consists of device-dependent counters and parameters.

The query_parms Parameter Block

For the CIO_QUERY operation, the *paramptr* parameter points to a **query_parms** structure. This structure is located in the **<sys/comio.h>** header file and contains the following fields:

status Contains additional information about the completion of the status block. Besides the status codes listed here, device-dependent codes may be returned:

CIO_OK	Indicates that the operation was successful.
CIO_INV_CMD	Indicates that an invalid command was issued.

bufptr	Points to the buffer where the statistic counter are to be copied.
buflen	Indicates the length of the buffer pointed to by the **bufptr** field.
clearall	When set to CIO_QUERY_CLEAR, the statistics counters are set to zero upon return.

Execution Environment

A CIO_QUERY operation can be called from the process environment only.

Return Values

In general, communication device-handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the CIO_QUERY operation are the following:

ENXIO	Indicates an attempt to use unconfigured device.
EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates an invalid parameter.
EIO	Indicates that an error has occurred.
ENOMEM	Indicates that the operation was unable to allocate the required memory.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENODEV	Indicates that the device does not exist.

Related Information

The **ddioctl** entry point, **ddmpx** entry point.

ddopen (Kernel Mode) Communications PDH Entry Point

Purpose

Performs data structure allocation and initialization for a communications Physical Device Handler (PDH).

Syntax

#include <sys/device.h>
#include <sys/comio.h>

int ddopen (*devno*, *devflag*, *chan*, *extptr***)**
dev_t *devno*;
ulong *devflag*;
int *chan*;
struct kopen_ext **extptr*;

Parameters for Kernel-Mode Processes

devno	Specifies the major and minor device numbers.
devflag	Specifies the flag word with the following definitions:

DKERNEL	This flag must be set for a calling a kernel-mode process.
DNDELAY	When set, the PDH performs nonblocking writes for this channel. Otherwise, blocking writes are performed.

chan	Specifies the channel number assigned by the device handler's **ddmpx** entry point.
extptr	Points to the **kopen_ext** structure.

Description

The **ddopen** entry point performs data structure allocation and initialization. Hardware initialization and other time consuming activities, such as call initialization, are not performed. This call is synchronous, which means it does not return until the **ddopen** entry point is complete.

The kopen_ext Parameter Block

For a kernel-mode process, the *extptr* parameter points to a **kopen_ext** structure. This structure contains the following fields:

status The status field may contain additional information about the completion of an open. Besides the status code listed here, device-dependent codes can be returned.

CIO_OK	Indicates that the operation was successful.
CIO_NOMBUF	Indicates that the operation was unable to allocate **mbuf** structures.
CIO_BAD_RANGE	Indicates an invalid address or parameter was specified.
CIO_HARD_FAIL	A hardware failure has been detected.

rx_fn
Specifies the address of a kernel procedure. The PDH calls this procedure whenever there is a receive frame to be processed. The **rx_fn** procedure must have the following syntax:

#include <sys/comio.h>

void rx_fn (*open_id, rd_ext_p, mbufptr***)**
ulong *open_id*;
struct read_extension **rd_ext_p*;
struct mbuf **mbufptr*;

open_id	Identifies the instance of open. It is passed to the PDH with the **ddopen** entry point.
rd_ext_p	Points to the read extension as defined in the **<sys/comio.h>** header file.
mbufptr	Points to an **mbuf** chain containing received data.

The kernel procedure calling the **ddopen** entry point is responsible for pinning the **rx_fn** kernel procedure before making the open call. It is the responsibility of code scheduled by the **rx_fn** procedure to free the **mbuf** chain.

tx_fn
Specifies the address of a kernel procedure. The PDH calls this procedure when the following sequence of events occurs:

1. The **DNDELAY** flag is set (determined by its setting in the last **uiop–>uio_fmode** field).
2. The most recent **ddwrite** entry point for this channel failed with a return code of EAGAIN.
3. Transmit queue for this channel now has room for a write.

The **tx_fn** procedure must have the following syntax:

#include <sys/comio.h>

void tx_fn (*open_id***)**
ulong *open_id*;

open_id	Identifier of instance of open. It is passed to the PDH with the **ddopen** call.

The kernel procedure calling the **ddopen** entry point is responsible for pinning the **tx_fn** kernel procedure before making the call.

stat_fn
Specifies the address of a kernel procedure to be called by the PDH whenever a status block becomes available. This procedure must have the following syntax:

#include <sys/comio.h>

void stat_fn (*open_id, sblk_ptr***)**;
ulong *open_id*;
struct status_block **sblk_ptr*

open_id	Identifier of instance of open. It is passed to the PDH with the **ddopen** entry point.
sblk_ptr	Points to a status block defined in the **sys/comio.h** file.

The kernel procedure calling the **ddopen** entry point is responsible for pinning the **stat_fn** kernel procedure before making the open call.

All three procedures are made synchronously from the off-level portion of the PDH at high priority from the PDH. Therefore, the called kernel procedure must return quickly. Parameter blocks are passed by reference and are valid only for the call's duration. After return from this call, the parameter block should not be accessed.

Execution Environment

A **ddopen** (kernel mode) entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **ddopen** entry point are the following:

EINVAL	Indicates an invalid parameter.
EIO	Indicates an error has occurred. The *status* field contains the relevant exception code.
ENODEV	Indicates that there is no such device.
EBUSY	Indicates that the maximum number of opens was exceeded or that the device was opened in exclusive-use mode.
ENOMEN	Indicates that the PDH was unable to allocate the space that it needed.
ENXIO	Indicates that an attempt was made to open the PDH before it was configured.
ENOTREADY	Indicates that the PDH is in the process of shutting down the adapter.

Related Information

The **ddopen** entry point for user-mode processes, **ddwrite** entry point, **ddclose** entry point, **ddmpx** entry point.

Use of the **mbuf** Structure in the Communications PDH, Status Blocks for the Communication Device Handlers.

ddopen (User Mode) Communications PDH Entry Point

Purpose

Performs data structure allocation and initialization for a communications PDH.

Syntax

#include <sys/device.h>
#include <sys/comio.h>

int ddopen (*devno*, *devflag*, *chan*, *ext*)
dev_t *devno*;
ulong *devflag*;
int *chan*;
int *ext*;

Parameters for User-Mode Processes

devno	Specifies the major and minor device numbers.
devflag	Specifies the flag word with the following definitions:

DKERNEL	This flag must be clear, indicating call by a user-mode process.
DNDELAY	If this flag is set, the physical device handler (PDH) performs nonblocking reads and writes for this channel. Otherwise, blocking reads and writes are performed for this channel.

chan	Specifies the channel number assigned by the device handler's **ddmpx** entry point.
ext	Device-dependent.

Description

The **ddopen** entry point performs data structure allocation and initialization. Hardware initialization and other time consuming activities, such as call initialization, are not performed. This call is synchronous, which means it does not return until the open operation is complete.

Execution Environment

A **ddopen** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **ddopen** entry point are the following:

EINVAL	Indicates an invalid parameter.
ENODEV	Indicates that there is no such device.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENOMEN	Indicates that the PDH was unable to allocate needed space.
ENOTREADY	Indicates that the PDH is in the process of shutting down the adapter.
ENXIO	Indicates that an attempt was made to open the PDH before it was configured.

Related Information

The **ddopen** entry point for kernel-mode processes, **ddclose** entry point.

ddread Communications PDH Entry Point

Purpose

Returns a data message to a user-mode process.

Syntax

#include <sys/device.h>
#include <sys/comio.h>

int ddread (*devno*, *uiop*, *chan*, *extptr*)
dev_t *devno*;
struct uio **uiop*;
int *chan*;
read_extension **extptr*;

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure. For a calling user-mode process, the **uio** structure specifies the location and length of the caller's data area in which to transfer information.
chan	Specifies the channel number assigned by the device handler's **ddmpx** entry point.
extptr	Can be NULL or else point to the **read_extension** structure. This structure is defined in the **<sys/comio.h>** header file.

Description

Note: The entry point should not to be called by a kernel-mode process.

The **ddread** entry point returns a data message to a user-mode process. This entry point may or may not block, depending on the setting of the **DNDELAY** flag. If a nonblocking read is issued and no data is available, the **ddread** entry point returns immediately with 0 (zero) bytes.

For this entry point, the *extptr* parameter may point to a user-supplied **read_extension** structure. This structure contains the following fields:

status The status value may contain additional information about the completion of the **ddread** entry point. Besides the status codes listed here, device-dependent codes can be returned:

CIO_OK	Indicates that the operation was successful.
CIO_BUF_OVFLW	Indicates that the frame was too large to fit in the receive buffer. The PDH truncates the frame and places the result in the receive buffer.

netid	Specifies the network ID associated with the returned frame. If a CIO_BUF_OVFLW was received, this field may be empty.
sessid	Specifies the session ID associated with the returned frame. If a CIO_BUF_OVFLW was received, this field may be empty.

Execution Environment

A **ddread** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **ddread** entry point are the following:

ENXIO	Indicates an attempt to use an unconfigured device.
EINVAL	Indicates an invalid parameter.
EIO	Indicates that an error has occurred.
EACCES	Indicates an invalid call from a kernel process.
EMSGSIZE	Indicates that the frame was too large to fit into the receive buffer and that no *extptr* parameter was supplied to provide an alternate means of reporting this error with a status of CIO_BUF_OVFLW.
EINTR	Indicates that a Locking mode sleep was interrupted.
EFAULT	Indicates that an invalid address was supplied.
EBIDEV	Indicates that the device specified does not exist.

Related Information

The **ddmpx** entry point, **ddwrite** entry point.

The CIO_START operation.

List of Common Status/Exception Codes.

Physical Device Handler Model Overview.

ddselect Communications PDH Entry Point

Purpose

Checks to see if a specified event or events has occurred on the device.

Syntax

#include <sys/device.h>
#include <sys/comio.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 conditions to be checked. The conditions are denoted by the bitwise OR of one or more of the following:

POLLIN	Check if receive data is available.
POLLOUT	Check if transmit available.
POLLPRI	Check if status is available.
POLLSYNC	Check if asychronous notification is available.

reventp	Points to the result of condition checks. A bitwise OR of the following conditions is returned:

POLLIN	Receive data is available.
POLLOUT	Transmit available.
POLLPRI	Status is available.

chan	Specifies the channel number assigned by the device handler's **ddmpx** entry point.

Description

Note: This entry point should not be called by a kernel-mode process.

The **ddselect** entry point checks and returns a status of 1 (one) or more conditions for a user-mode process. It works the same way The common **ddselect** entry point does.

Execution Environment

A **ddselect** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **ddselect** entry point are the following:

ENXIO	Indicates an attempt to use an unconfigured device.
EINVAL	Indicates that an invalid argument was specified.
EACCES	Indicates an invalid call from a kernel process.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENODEV	Indicates that the device does not exist.

Related Information

The **ddmpx** entry point.

ddwrite Communications PDH Entry Point

Purpose

Queues a message for transmission or blocks until the message can be queued.

Syntax

#include <sys/device.h>
#include <sys/comio.h>

int ddwrite (*devno*, *uiop*, *chan*, *extptr***)**
dev_t *devno*;
struct uio **uiop*;
int *chan*;
struct write_extension **extptr*;

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure specifying the location and length of the caller's data.
chan	Specifies the channel number assigned by the device handler's **ddmpx** entry point.
extptr	Points to a **write_extension** structure. If the *extptr* parameter is NULL, then default values are assumed.

Description

The **ddwrite** entry point either queues a message for transmission or blocks until the message can be queued, depending upon the setting of the **DNDELAY** flag.

The **ddwrite** routine is responsible for determining whether the data is in user or system space by looking at the **uiop–>uio_segflg** field. If the data is in system space, then the **uiop–>uio_iov–>iov_base** field contains an **mbuf** pointer. The **mbuf** chain contains the data for transmission. The **uiop–>uio_resid** field has a value of 4. If the data is in user space, the data is located in the same manner as for the **ddwrite** entry point.

The write_extension Parameter Block

For this entry point, the *extptr* parameter can point to a **write_extension** structure. This structure is defined in the **<sys/comio.h>** header file and contains the following fields:

status Indicates the status of the port. This field may contain additional information about the completion of the **ddwrite** entry point. Besides the status codes listed here, device dependent codes can be returned:

CIO_OK	Indicates that the operation was successful.
CIO_NOMBUF	Indicates that the operation was unable to allocate **mbuf** structures.

flag	Contains a bitwise OR of one or more of the following: **CIO_NOFREE_MBUF** Requests that the PDH not free the **mbuf** structure after transmission is complete. The default is bit clear (free the buffer). For a user-mode process, the PDH always frees the **mbuf** structure. **CIO_ACK_TX_DONE** Requests that, when done with this operation, the PDH acknowledges completion by building a CIO_TX_DONE status block. In addition, requests the PDH either call the kernel status function or (for a user-mode process) place the status block in the status/exception queue. The default is bit clear (do not acknowledge transmit completion).
writid	Contains the write ID to be returned in the CIO_TX_DONE status block. This field is ignored if the user did not request transmit acknowledgment by setting CIO_ACK_TX_DONE in the **flag** field.
netid	May contain the network ID.

Execution Environment

A **ddwrite** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **ddwrite** entry point are the following:

ENXIO	Indicates an attempt to use an unconfigured device.
EINVAL	Indicates an invalid parameter.
EAGAIN	Indicates that the transmit queue is full and the **DNDELAY** flag is set. The command was not accepted.
EFAULT	Indicates that an invalid address was specified.
EINTR	Indicates that a Blocking mode sleep was interrupted.
ENOMEM	Indicates that the operation was unable to allocate the needed **mbuf** space.
ENOCONNECT	Indicates that a connection was not established.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENODEV	Indicates that the device does not exist.

Related Information

The **ddmpx** entry point.

The CIO_START operation, CIO_GET_STAT operation.

entclose Ethernet Device Handler Entry Point

Purpose

Resets the Ethernet device to a known state and returns system resources to the system.

Syntax

#include <sys/device.h>

int entclose (*devno*, *chan*, *ext*)
dev_t *devno*;
int *chan*, *ext*;

Parameters

devno	Identifies the major and minor device numbers.
chan	Specifies the channel number assigned by the **entmpx** entry point.
ext	Ignored by the Ethernet device handler.

Description

The **entclose** entry point closes the device. It is called when a user-mode caller issues a **close** subroutine. Before issuing the **entclose** entry point, the caller should have issued a CIO_HALT operation for each successfully issued CIO_START operation during the particular instance of the open.

Note: For each **entopen** entry point issued, there must be a corresponding **entclose** entry point.

If the caller has specified a multicast address, the caller needs to issue the appropriate **entioctl** operation to remove all multicast addresses before issuing the **entclose** entry point.

Execution Environment

An **entclose** entry point can be called from the process environment only.

Return Values

ENXIO	Indicates that the device is not configured.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENODEV	Indicates that the specified device does not exist.

Implementation Specifics

The **entclose** entry point functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entopen** entry point, **entmpx** entry point.

The CIO_START operation.

The **close** subroutine.

entconfig Ethernet Device Handler Entry Point

Purpose

Provides functions for initializing, terminating, and querying the vital product data (VPD) of the Ethernet device handler.

Syntax

#include <sys/device.h>
#include <sys/uio.h>

int entconfig (*devno*, *cmd*, *uiop***)**
dev_t *devno*;
int *cmd*;
struct uio **uiop*;

Parameters

devno	Specifies the major and minor device numbers.
cmd	Specifies which function this routine should performed. There are three possible functions:
	CFG_INIT Initializes device handler and internal data areas.
	CFG_TERM Terminates the device handler.
	CFG_QVPD Queries VPD.
uiop	Points to a **uio** structure. The **uio** structure is defined in the **<sys/uio.h>** file.

Description

The **entconfig** entry point provides functions for initializing, terminating, and querying the VPD of the Ethernet device handler. The following are three possible **entconfig** operations:

- CFG_INIT

  For the CFG_INIT operation, the Ethernet device handler registers its entry points by placing them into the device switch table for the major device number specified by the *devno* parameter. The **uio** structure contains the *iov_base* pointer, which points to the Ethernet device dependent structure (DDS). The caller provides the **uio** structure. The structure is copied into an internal save area by the **init** function.

- CFG_TERM

  For the CFG_TERM operation, if there are no outstanding opens, the following occurs:

  – The Ethernet device handler marks itself terminated and prevents subsequent opens.

  – All dynamically allocated areas are freed.

  – All Ethernet device handler entry points are removed from the device switch table.

- CFG_QVPD

  The CFG_QVPD operation returns the Ethernet VPD to the caller. The VPD is placed in the area specified by the caller in the **uio** structure.

Execution Environment

An **entconfig** entry point can be called from the process environment only.

Return Values

EINVAL	Indicates an invalid address range or opcode (common to all three operations).
EBUSY	Indicates that the device was already open in Diagnostic Mode and so the open request was denied (common to the CFG_TERM and CFG_INIT operations).
EEXIST	Indicates that the DDS structure already exists (CFG_TERM operation).
ENODEV	Indicates that no such device exists (common to all three operations.).
EUNATCH	Indicates that the protocol driver was not attached (common to the CFG_TERM operation).
EFAULT	Indicates that an invalid address was specified (common to the CFG_QVPD and CFG_INIT operations).
EINVAL	Indicates an invalid range or opcode (common to all three operations).
EACCES	Indicates that permission was denied because the device was already open, or because there were outstanding opens that were unable to terminate (common to the CFG_TERM and CFG_QVPD operations).
ENOENT	Indicates that there is no DDS to delete (common to the CFG_TERM and CFG_QVPD operations).
ENXIO	Indicates that no such device exists or that the maximum number of adapters was exceeded (common to all three operations).
EEXIST	Indicates that the DDS structure already exists (common to the CFG_TERM and CFG_INIT operations).
EFAULT	Indicates that an invalid address was specified (common to the CFG_TERM and CFG_INIT operations).
ENOMEM	Indicates insufficient memory (common to the CFG_INIT operation).

Implementation Specifics

The **entconfig** entry point functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **uio** structure.

Device-Dependent Structure in *Kernel Extensions and Device Support Programming Concepts.*

entioctl Ethernet Device Handler Entry Point

Purpose

Provides various functions for controlling the Ethernet device.

Syntax

#include <sys/device.h>
#include <sys/devinfo.h>
#include <sys/ioctl.h>
#include <sys/comio.h>
#include <sys/entuser.h>

int entioctl (*devno*, *cmd*, *arg*, *devflag*, *chan*, *ext***)**
dev_t *devno*;
int *cmd*, *arg*;
ulong *devflag*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
cmd	Specifies which operation is to be performed. The possible **entioctl** operation codes can be found in the **<sys/ioctl.h>** and **sys/comio.h** files.
arg	Specifies the address of the **entioctl** parameter block.
devflag	This parameter is ignored by the Ethernet device handler.
chan	Specifies the channel number assigned by the **entmpx** routine.
ext	This parameter is not used by the Ethernet device handler.

Description

The **entioctl** Ethernet device handler entry point provides various functions for controlling the Ethernet device. There are eight common **entioctl** operations plus four additional **entioctl** operations available for diagnostic purposes.

These are the eight valid **entioctl** operations:

CIO_START	Starts a session and registers a network ID.
CIO_HALT	Halts a session and removes the registered network ID.
CIO_QUERY	Returns the current RAS counter values.
CIO_GET_STAT	Returns the current adapter and device handler status
ENT_SET_MULTI	Sets or resets a multicast address.
IOCINFO	Returns I/O character information.
CCC_GET_VPD	Returns vital product data (VPD) about the adapter.
ENT_SET_MULTI	Sets the multicast address for the Ethernet device.

These are the four valid **entioctl** operations for diagnostic purposes:

CCC_TRCTBL	Returns the address of the internal device driver trace table.
CIO_POS_ACC	Reads or writes a byte from or to a selected adapter POS register
CIO_REG_ACC	Reads or writes a byte from or to a selected adapter I/O register.
CIO_MEM_ACC	Reads or writes data from or to selected adapter RAM addresses.

The following are DMA Facilities Operations:

ENT_LOCK_DMA	Sets up (locks) a user buffer to DMA from or to the adapter.
ENT_UNLOCK_DMA	Clears (unlocks) a user buffer from DMA control.

Execution Environment

An **entioctl** entry point can be called from the process environment only.

Implementation Specifics

The **entioctl** entry point functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entmpx** entry point.

CCC_GET_VPD entioctl Operation (Query Vital Product Data)

Purpose

Returns vital product data (VPD) about the Ethernet adapter.

Description

The CCC_GET_VPD operation returns vital product data (VPD) about the Ethernet adapter. For this operation, the *arg* parameter points to the **vital_product_data** structure. This structure is defined in the **<sys/ciouser.h>** file and has the following fields:

status Indicates the status of the VPD characters returned in the array of characters. Valid values for this status word are found in the **<sys/ciouser.h>** header file:

VPD_NOT_READ	VPD data has not been obtained from the adapter.
VPD_NOT_AVAIL	VPD data is not available for this adapter.
VPD_INVALID	VPD data that was obtained is invalid.
VPD_VALID	VPD data was obtained and is valid.

length Specifies the number of bytes that are valid in the VPD character array. This value can be 0 (zero), depending on the status returned.

vpd[*n*] An array of characters that contain the adapter's VPD. The number of valid characters is determined by the length value.

Execution Environment

A CCC_GET_VPD operation can be called from the process environment only.

Return Values

EFAULT	Indicates an invalid address.
ENXIO	Indicates that no such device exists.

Implementation Specifics

The CCC_GET_VPD operation functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entioctl** entry point.

The Ethernet Vital Product Data Structure.

CIO_GET_STAT entioctl Operation (Get Status)

Purpose

Returns the current Ethernet adapter and device handler status.

Description

Note: Only user-mode callers can use the CIO_GET_STAT operation.

The CIO_GET_STAT operation returns the current Ethernet adapter and device handler status. The device handler fills in the parameter block with the appropriate information upon return. For this operations, the *arg* parameter points to a status block structure. This structure is defined in the **<sys/comio.h>** header file.

There are two Ethernet-specific status blocks:

- CIO_START_DONE
- CIO_HALT_DONE.

The Ethernet device handler also returns the following general communications status blocks:

- CIO_LOST_STATUS
- CIO_NULL_BLK
- CIO_TX_DONE
- CIO_ASYNC_STATUS.

Status Blocks for the Ethernet Device Handler

Status blocks are used to communicate status and exception information to user-mode processes.

User-mode processes receive a status block whenever they request a CIO_GET_STAT operation. A user-mode process can wait for the next available status block by issuing a **entselect** entry point with the specified POLLPRI event.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, CIO_START_DONE). The following are the possible Ethernet status blocks:

There are two Ethernet specific status blocks:

- CIO_START_DONE
- CIO_HALT_DONE.

The Ethernet device handler also returns the following general communications status blocks:

- CIO_LOST_STATUS
- CIO_NULL_BLK
- CIO_TX_DONE
- CIO_ASYNC_STATUS.

CIO_START_DONE Status Block

On a successfully completed Start Device **entioctl** operation, the status block is filled in as follows:

option[0]	CIO_OK.
option[1]	The high-order 2 bytes are filled in with the high-order 2 bytes of the network address. The low-order 2 bytes are filled in with the middle two bytes of the network address.
option[2]	The low-order 2 bytes are filled in with the low-order 2 bytes of the network address.

CIO_HALT_DONE Status Block

On a successfully completed Halt Device **entioctl** operation, the status block is filled in as follows:

option[0]	CIO_OK
option[1]	Not used.
option[2]	Not used.

Execution Environment

A CIO_GET_STAT operation can be called from the process environment only.

Return Values

EACCES	Indicates that permission was denied.
EBUSY	Indicates that the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
ENODEV	Indicates that no such device exists.
ENXIO	Indicates that an attempt was made to use an unconfigured device.

Implementation Specifics

The CIO_GET_STAT operation functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entioctl** entry point.

Status Blocks for Communications Device Handlers.

CIO_HALT entioctl Operation (Halt Device)

Purpose

Ends a session with the Ethernet device handler.

Description

The CIO_HALT operation ends a session with the Ethernet device handler. The caller indicates the network ID to be halted. This CIO_HALT operation corresponds to the CIO_START operation that was successfully issued with the specified network ID.

Data for the specified network ID is no longer received. Data already received for the specified network ID, before the Halt Device operation, is still passed up to a user-mode caller. The **entselect** and **entread** entry points pass this data. Data is passed to a kernel-mode caller by the **rx_fn** routine specified at open time.

When a CIO_HALT operation has ended the last open session on a channel, the caller should issue the **entclose** operation.

Note: If the caller has specified a multicast address, the caller needs to issue the appropriate **entioctl** entry point to remove all the multicast addresses before issuing a CIO_HALT.

For CIO_HALT operation, the *arg* parameter points to a **session_blk** structure. This structure is defined in the **<sys/comio.h>** header file and contains the following fields:

status There are two possible returned status values:

- CIO_OK
- CIO_NETID_INV.

netid Specifies the network ID. When IEEE 802.3 Ethernet is being used, the network ID is placed in the least significant byte of the **netid** field.

Execution Environment

A CIO_HALT operation can be called from the process environment only.

Return Values

EINVAL	Indicates that the specified network ID is not in the table.
EBUSY	Indicates that the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
ENODEV	Indicates that no such device exists.
ENXIO	Indicates that an attempt was made to use an unconfigured device.

Implementation Specifics

The CIO_HALT operation functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entread** entry point, **entselect** entry point, **entioctl** entry point.

The CIO_START operation.

List of common status/exception codes.

CIO_QUERY entioctl Operation (Query Statistics)

Purpose

Reads the counter values accumulated by the Ethernet device handler.

Description

The CIO_QUERY operation is used by the caller to read the counter values accumulated by the device handler. The counters are initialized to 0 (zero) by the each CIO_START operation issued.

For the CIO_QUERY operation, the *arg* parameter points to a **query_parms** structure. This structure is defined in the **<sys/comio.h>** header file and contains the following fields:

status	Specifies the current status condition. This fields accepts two possible status values: • CIO_OK • COP_BUF_OVFLW.
buffptr	Specifies the address of a buffer where the returned statistics are to be placed.
bufflen	Specifies the length of the buffer.
clearall	When set to a value of CIO_QUERY_CLEAR, the counters are cleared upon completion of call. This value is defined in the **<sys/comio.h>** file.

The CIO_QUERY operation specifies the device-specific information placed in the supplied buffer. The counter placed in the supplied buffer by this operation is the **ent_query_stats_t** structure, which is defined in the **<sys/entuser.h>** file.

Execution Environment

A CIO_QUERY operation can be called from the process environment only.

Return Values

ENOMEM	Indicates insufficient memory.
EIO	Indicates that the caller's buffer is too small.
EBUSY	Indicates that the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
ENODEV	Indicates that no such device exists.
ENXIO	Indicates that an attempt was made to use an unconfigured device.

Implementation Specifics

The CIO_QUERY operation functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entopen** entry point, **entioctl** entry point.

The **CIO_START** operation.

CIO_START entioctl Operation (Start Device)

Purpose

Establishes a session with the Ethernet device handler.

Description

The CIO_START operation establishes a session with the Ethernet device handler. The caller notifies the device handler of the network ID that it will be using. The caller can issue multiple CIO_START operations. For each successful start issued, there should be a corresponding CIO_HALT operation issued.

If the CIO_START operation is the first issued, the device handler initializes and opens the Ethernet adapter. When the first CIO_START operation successfully completes, the adapter is ready to transmit and receive data. The Ethernet adapter can receive the following packet types:

- Packets that match the Ethernet adapter's burned-in address (or the address specified in the device dependent structure (DDS))
- Broadcast packets
- Multicast packets
- Packets that match the network ID specified in the **netid** field.

The Ethernet device handler allows a maximum of 32 network IDs. The network ID must correspond to the type field in a standard Ethernet packet or the destination service access point (DSAP) address in an IEEE 802.3 packet.

For the CIO_START operation, the *arg* parameter points to a **session_blk** structure. This structure is defined in the **<sys/comio.h>** header file and contains the following fields:

status There are four possible returned status values:

- CIO_OK
- CIO_NETID_FULL
- CIO_NETID_DUP
- CIO_HARD_FAIL.

netid Specifies the network ID the caller uses on the network. When IEEE 802.3 Ethernet is being used, the network ID is placed in the least significant byte of the **netid** field.

Note: The Ethernet device handler does not allow the caller to specify itself as the wild card network ID.

length This field is used to specify the number of valid bytes in the **netid** field for mixed Ethernet. Valid values are 1 or 2.

After the CIO_START operation has successfully completed, the caller can issue any valid Ethernet command.

Note: The Ethernet device handler does not support promiscuous addressing.

Execution Environment

A CIO_START operation can be called from the process environment only.

Return Values

ENETUNREACH	Indicates that the operation was unable to reach the network.
EBUSY	Indicates that the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
ENODEV	Indicates that no such device exists.
ENXIO	Indicates that there was an attempt to use an unconfigured device.
ENOSPC	Indicates that the **netid** table is full.
EADDRINUSE	Indicates a duplicate network ID.

Implementation Specifics

The CIO_START operation functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entioctl** entry point.

The CIO_HALT operation.

List of Common Status/Exception Codes.

ENT_SET_MULTI entioctl Operation (Set Multicast Address)

Purpose

Sets the multicast address for the Ethernet device.

Description

The ENT_SET_MULTI operation sets the multicast address for the Ethernet device. For this operation, the *arg* parameter points to the **ent_set_multi_t** structure. This structure is defined in the **<sys/entuser.h>** header file and contains the following fields:

opcode	Specifies whether to add or delete a multicast address. When this field is ENT_ADD, the multicast address is added to the multicast entry table. When this field is ENT_DEL, the multicast address is removed from the multicast entry table. Valid Ethernet types are defined in the **<sys/entuser.h>** header file.
multi_addr(6)	Identifies the multicast address array where **multi_addr(0)** is the most significant byte and **multi_addr(5)** is the least significant byte.

Note: The Ethernet device handler allows a maximum of 20 multicast addresses.

Execution Environment

An ENT_SET_MULTI operation can be called from the process environment only.

Return Values

EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates an invalid operation code.
ENOSPC	Indicates that no space was left on the device. (The multicast table is full.)
ENOTREADY	Indicates that the device was not ready. (The first CIO_START operatin was not issued and not completed.)
EACCES	Indicates that permission was denied. (The device was open in Diagnostic mode.)
EAFNOSUPPORT	Indicates that the address family was not supported by protocol. (The multicast bit in the address was not set.)
ENXIO	Indicates that no such device exists.

Implementation Specifics

The ENT_SET_MULTI operation functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entioctl** entry point.

The CIO_START operation.

IOCINFO entioctl Operation (Describe Device)

Purpose

Returns a structure that describes the Ethernet device.

Description

The IOCINFO operation returns a structure that describes the Ethernet device. For this operation, the *arg* parameter points to the **ethernet** substructure, which is defined as part of the **devinfo** structure. This **devinfo** structure is located in the **<sys/devinfo.h>** file and contains the following fields:

devtype	Identifies the device type. The Ethernet type is DD_NET_DH. This label is defined in the **<sys/devinfo.h>** header file.
devsubtype	Identifies the device subtype. The Ethernet subtype is DD_EN. This label can be found in the **<sys/devinfo.h>** header file.
broad_wrap	Indicates the adapter's ability to receive its own packets. A value of 1 (one) indicates that the adapter can receive its own packets. A value of 0 (zero) indicates that the adapter cannot receive its own packets. For this adapter, a value of 0 (zero) is returned.
rdto	Specifies the receive data transfer offset. This value indicates an offset (in bytes) into the data area of the receive page-sized **mbuf** structure. The device handler places received data in this buffer.
haddr	Identifies the 6-byte unique Ethernet adapter address. This address is the burned-in address that is readable from the adapter's vital product data (VPD). The most significant byte of the address is placed in the **haddr(0)** field. The least significant byte is placed in the address specified by the **haddr(5)** field.
net_addr	Identifies the 6-byte unique Ethernet adapter address that is currently being used by the Ethernet adapter card. This address is either the burned–in address (readable from the VPD) or the alternate address that can be used to configure the adapter. The most significant byte of the address is placed in the address specified by the **net_addr(0)** field. The least significant byte is placed in the address specified by **net_addr(5)** field.

The parameter block is filled in with the appropriate values upon return.

Execution Environment

An IOCINFO operation can be called from the process environment only.

Return Values

EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates an invalid operation code.
ENXIO	Indicates that no such device exists.

Implementation Specifics

The IOCINFO operation functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entioctl** entry point.

entmpx Ethernet Device Handler Entry Point

Purpose

Provides allocation and deallocation of a channel for an Ethernet device handler.

Syntax

#include <sys/device.h>

int entmpx (*devno*, *chanp*, *channame*)
dev_t *devno*;
int **chanp*;
char **channame*;

Parameters

devno	Specifies the major and minor device numbers.
chanp	Contains the channel ID passed as a reference parameter. If the *channame* parameter is NULL, this parameter is the ID of the channel to be deallocated. Otherwise, the *chanp* parameter is set to the ID of the allocated channel.
channame	Points to the remaining path name describing the channel to be allocated. This field points to one of the following values:

NULL	The channel is to be deallocated.
A pointer to NULL string	Indicates the channel ID generated by the **entmpx** routine and allows a normal open sequence of the Ethernet device.
A pointer to D	The channel ID generated by the **entmpx** routine. This ID allows the Ethernet device to be opened in Diagnostic mode.

Description

The **entmpx** entry point provides allocation and deallocation of a channel for an Ethernet device handler. It is not called directly by a user of the Ethernet device handler. The kernel calls the **entmpx** entry point in response to an open or close request.

Note: If the Ethernet device has been successfully opened, any subsequent Diagnostic mode open requests fails. If the device has been successfully opened in Diagnostic mode, all subsequent open requests fail.

Execution Environment

An **entmpx** entry point can be called from the process environment only.

Return Values

EBUSY	Indicates that the maximum number of opens was exceeded.
ENOMSG	No message of desired type.
ENODEV	Indicates that the specified device does not exist.
ENXIO	Indicates that the device is not configured.

Implementation Specifics

The **entmpx** entry point functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entopen** entry point.

entopen Ethernet Device Handler Entry Point

Purpose

Initializes the Ethernet device handler and allocates the required system resources.

Syntax

#include <sys/device.h>
#include <sys/comio.h>
#include <sys/entuser.h>

int entopen (*devno*, *devflag*, *chan*, *ext*)
dev_t *devno*;
ulong *devflag*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers for both kernel- and user-mode entry pointers.
devflag	Specifies the **DKERNEL** flag, which must be set for a kernel-mode entry pointer. This flag cannot be set for user-mode entry pointers.
chan	Specifies the channel number assigned by the **entmpx** routine for both kernel- and user-mode entry pointers.
ext	Points to a **kopen_ext** structure. This structure is defined in the **<sys/comio.h>** header file. This parameter is valid only for kernel-mode users; it is NULL for user-mode users.

Description

The **entopen** entry point prepares the Ethernet device for transmitting and receiving data. It is called when a user-mode entry pointer issues an **open**, **openx**, or **creat** subroutine. After the **entopen** entry point has successfully completed, the entry pointer must issue a CIO_START operation before using the Ethernet device handler. The device handler is then opened for reading and writing data.

Execution Environment

An **entopen** entry point can be called from the process environment only.

Return Values

EINVAL	Indicates an invalid range, opcode, or the device is not in Diagnostic mode.
ENOMEM	Indicates insufficient memory.
ENOTREADY	Indicates that the device was not ready. The first CIO_START operation was not issued and hence not completed.
ENXIO	Indicates that no such device exists. (The maximum number of adapters was exceeded.)

Implementation Specifics

The **entopen** entry point functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entmpx** entry point, **entclose** entry point.

The CIO_START operation.

The**creat** subroutine, **open** subroutine, **openx** subroutine.

entread Ethernet Device Handler Entry Point

Purpose

Provides the means of receiving data from the Ethernet device handler.

Syntax

#include <sys/device.h>
#include <sys/uio.h>

int entread (*devno*, *uiop*, *chan*, *ext*)
dev_t *devno*;
struct uio **uiop*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure. This structure is defined in the **<sys/uio.h>** header file.
chan	Specifies the channel number assigned by the **entmpx** routine.
ext	Can specify the address of the **entread** parameter block. If the *ext* parameter is NULL, then no parameter block is specified.

Description

Note: The **entread** entry point should only be called by user-mode callers.

The **entread** entry point provides the means of receiving data from the Ethernet device handler. When a user-mode caller issues a **read, readx**, **readv**, or **readvx** subroutine, the kernel calls the **entread** entry point.

When the **entread** entry point is called, the file system fills in the **uio** structure fields with the appropriate values. In addition, the device handler copies the data into the buffer specified by the caller.

For the **entread** entry point, the *ext* parameter may point to the **read_extension** structure. This structure is defined in the **<sys/comio.h>** header file and contains the following field:

status Contains a status code. This field may be one of the following values:

- CIO_OK
- CIO_BUF_OVRFLW
- CIO_NOT_STARTED.

Execution Environment

An **entread** entry point can be called from the process environment only.

Return Values

EACCES	Indicates that permission was denied because the device was already open. Diagnostic mode open request denied.
EFAULT	Indicates that an invalid address was specified.
EINTR	Indicates that interrupted system call.
EIO	Indicates an I/O error.
EMSGSIZE	Indicates that the data returned was too large for the buffer.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENODEV	Indicates that the specified device does not exist.
ENOCONNECT	Indicates that no connection was established.
ENXIO	Indicates that an attempt was made to use an unconfigured device.

Implementation Specifics

The **entread** entry point functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entmpx** entry point.

List of Common Status/Exception Codes.

entselect Ethernet Device Handler Entry Point

Purpose

Determines if a specified event has occurred on the Ethernet device.

Syntax

#include <sys/device.h>
#include <sys/comio.h>

int entselect (*devno*, *events*, *reventp*, *chan***)**
dev_t *devno*;
ushort *events*;
ushort **reventp*;
int *chan*;

Parameters

devno	Specifies the major and minor device numbers.
events	Identifies the events that are to be checked.
reventp	Returned events pointer passed by reference. This pointer is used by the **entselect** entry point to indicate which of the selected events are true when the call occurs.
chan	Specifies the channel number assigned by the **entmpx** entry point.

Description

Note: Only user-mode callers should use the **entselect** entry point.

The **entselect** entry point determines if a specified event has occurred on the Ethernet device. This entry point must be called with the **select** or **poll** subroutine.

When the Ethernet device handler is in a state in which the specified event can never be satisfied (for example, an adapter failure), then the **entselect** entry point sets the returned event flags to a 1 (one). This prevents the **select** or **poll** subroutine from waiting indefinitely.

Execution Environment

An **entselect** entry point can be called from the process environment only.

Return Values

EACCES	Indicates that permission was denied because the device had not been initialized. Indicates that the Diagnostic mode open request was denied. Indicates that permission was denied because the call is from a kernel-mode process.
ENXIO	Indicates that there was no such device. (Maximum number of adapters was exceeded.)
EBUSY	Indicates that the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
ENODEV	Indicates that no such device exists.

Implementation Specifics

The **entselect** entry point functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entmpx** entry point.

The **select** subroutine, **poll** subroutine.

entwrite Ethernet Device Handler Entry Point

Purpose

Provides the means for transmitting data from the Ethernet device.

Syntax

#include <sys/device.h>
#include <sys/uio.h>
#include <sys/comio.h>
#include <sys/entuser.h>

int entwrite (*devno*, *uiop*, *chan*, *ext*)
dev_t *devno*;
struct uio **uiop*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure that provides variables to control the data transfer operation. This **uio** structure is defined in the **<sys/uio.h>** file.
chan	Specifies the channel number assigned by the **entmpx** entry point.
ext	Specifies the address of the **entwrite** parameter block. If the *ext* parameter is NULL, then no parameter block is specified.

Description

The **entwrite** entry point provides the means for transmitting data for the Ethernet device. The kernel calls it when a user-mode caller issues a **write, writex, writev**, or **writevx** subroutine.

For a user-mode caller, the file system fills in the **uio** structure variables with the appropriate values. A kernel-mode caller must fill in the **uio** structure in the same manner as the general **ddwrite** entry point.

The write_extension Parameter Block

For the **entwrite** entry point, the *ext* parameter is a pointer to a **write_extension** structure. This structure is defined in the **<sys/comio.h>** header file and contains the following fields:

status
: Identifies the status of the write operation. This field is in the **write_extension** structure and accepts the following values:

- CIO_OK
- CIO_TX_FULL.

write_id
: For a user-mode caller, the **write_id** field is returned to the caller by the CIO_GET_STAT operation if the ACK_TX_DONE option is selected. For a kernel-mode caller, the **write_id** field is returned to the caller by the **stat_fn** function that was provided at open time.

Execution Environment

An **entwrite** entry point can be called from the process environment only.

Return Values

EAGAIN	Indicates that the transmit queue is full.
EFAULT	Indicates that an invalid address was specified.
EINTR	Indicates an interrupted system call.
EINVAL	Indicates an invalid range or opcode.
ENOCONNECT	Indicates that no connection was established.
ENOMEM	Indicates insufficient memory.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENODEV	Indicates that the specified device does not exist.
ENXIO	Indicates that an attempt was made to use an unconfigured device.

Implementation Specifics

The **entwrite** entry point functions with a Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **entread** entry point, **entmpx** entry point.

The **uio** structure.

The **write**, **writex**, **writev**, or **writevx** subroutine.

mpclose Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Resets the Multiprotocol (MPQP) device to a known state and returns system resources back to the system on the last close for that adapter.

Syntax

int mpclose (*devno*, *chan*, *ext*)
dev_t *devno*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
chan	Specifies the channel number assigned by the **mpmpx** entry point.
ext	Ignored by the MPQP device handler.

Description

The **mpclose** entry point routine resets the MPQP device to a known state and returns system resources to the system on the last close for that adapter. The port no longer accepts read, write, or ioctl operation requests. The **mpclose** entry point is called in user mode by issuing a **close** system call. The **mpclose** entry point is invoked in response to a **fp_close** kernel service.

On an **mpclose** entry point, the MPQP device handler does the following:

- Frees all internal data areas for the corresponding **mpopen** entry point
- Purges receive data queued for this **mpopen** entry point.

On the last **mpclose** entry point for a particular adapter, the MPQP device handler also does the following:

- Frees its interrupt level to the system
- Frees the DMA channel
- Disables adapter interrupts
- Sets all internal data elements to their default settings.

The **mpclose** entry point closes the device. For each **mpopen** entry point issued, there must be a corresponding **mpclose** entry point.

Before issuing the **mpclose** entry point, the caller should issue a CIO_HALT operation for each CIO_START operation issued during that particular instance of open. If a close request is received without a preceding CIO_HALT operation, the functions of the halt are performed. This should only occur during abnormal termination of the port.

Execution Environment

The **mpclose** entry point can be called from the process environment only.

Return Values

ECHRNG	Indicates that the channel number is too large.
ENXIO	Indicates that the port intialization failed. This code could also indicate that the registration of the interrupt failed.
ECHRNG	Indicates that the channel number is out of range (too high).

Implementation Specifics

The **mpclose** entry point functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **mpioctl** entry point, **mpopen** entry point.

The **fp_close** kernel service.

The CIO_HALT operation, CIO_START operation.

The **close** system call.

mpconfig Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Provides functions for initializing and terminating the Multiprotocol (MPQP) device handler and adapter.

Syntax

#include <sys/uio.h>

int mpconfig (*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 this routine. There are two possible functions:
	CFG_INIT Initializes device handler and internal data areas.
	CFG_TERM Terminates the device handler.
uiop	Points to a **uio** structure. The **uio** structure is defined in the **<sys/uio.h>** header file.

Description

The **mpconfig** entry point provides functions for initializing and terminating the MPQP device handler and adapter. It is invoked through the **/sys/config** device driver at device configuration time. This entry point supports the following operations:

- **CFG_INIT**
- **CFG_TERM.**

Execution Environment

The **mpconfig** entry point can be called from the process environment only.

Implementation Specifics

The **mpconfig** entry point functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **ddconfig** (CFG_INIT) routine, **ddconfig** (CFG_TERM) routine.

mpioctl Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Provides various functions for controlling the Multiprotocol (MPQP) device.

Syntax

#include <sys/devinfo.h>
#include <sys/ioctl.h>
#include <sys/comio.h>
#include <sys/mpqp.h>

int mpioctl (*devno*, *cmd*, *extptr*, *devflag*, *chan*, *ext*)
dev_t *devno*;
int *cmd*, *extptr*;
ulong *devflag*
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
cmd	Identifies the operation to be performed.
extptr	Specifies an address of the parameter block.
devflag	This allows **mpioctl** calls to inherit properties that were specified at open time. The MPQP device handler inspects the **DNDELAY** flag for **ioctl** calls. Kernel-mode data link control (DLC), also sets the **DKERNAL** flag must be set.
chan	Specifies the channel number assigned by the **mpmpx** entry point.
ext	Not used by MPQP device handler.

Description

The **mpioctl** MPQP device handler entry point provides various functions for controlling the MPQP device. There are 7 valid **mpioctl** operations:

mpioctl MPQP Operations

CIO_START	Initiates a session with the MPQP device handler.
CIO_HALT	Ends a session with the MPQP device handler.
CIO_QUERY	Reads the counter values accumulated by the MPQP device handler.
CIO_GET_STATUS	Gets the status of the current MPQP adapter and device handler.
MP_START_AR	Puts the MPQP port into Autoresponse mode.
MP_STOP_AR	Permits the MPQP port to exit Autoresponse mode.
MP_CHG_PARMS	Permits the DLC to change certain profile parameters after the MPQP device has been started.

The possible **mpioctl** operation codes can be found in the **<sys/ioctl.h>**, **<sys/comio.h>**, and **<sys/mpqp.h>** header files.

Execution Environment

The **mpioctl** entry point can be called from the process environment only.

Return Values

ENXIO	Indicates the adapter number is out of range.
ENOMEM	Indicates the no **mbufs** or **mbuf** clusters are available.

Implementation Specifics

The **mpioctl** entry point functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **mpmpx** entry point.

CIO_GET_STAT mpioctl Operation (Get Status)

Purpose

Gets the status of the current Multiprotocol (MPQP) adapter and device handler.

Description

Note: Only user-mode processes can use the CIO_GET_STAT operation.

The CIO_GET_STAT operation gets the status of the current MPQP adapter and device handler. For the MPQP device handler, both solicited and unsolicited status can be returned.

Solicited status is status information that is returned as a completion status to a particular operation. The CIO_START, CIO_HALT, and **mpwrite** operations all have solicited status returned. However, for many asynchronous events that are common to wide-area networks, these are considered unsolicited status. The asynchronous events are divided into three classes:

- Hard failures
- Soft failures
- Informational (or status-related) messages.

For the CIO_GET_STAT operation, the *extptr* parameter points to a **status_block** structure. This structure is filled with the appropriate information by the device handler upon return. The **status_block** structure is defined in the **<sys/comio.h>** header file and returns one of seven possible status conditions:

- CIO_START_DONE
- CIO_HALT_DONE
- CIO_TX_DONE
- CIO_ASYNC_STATUS
- MP_RDY_FOR_MAN_DIAL
- MP_ERR_THRESHLD_EXC.

Status Blocks for the Multiprotocol Device Handler

Status blocks are used to communicate status and exception information to user-mode processes.

User-mode processes receive a status block whenever they request a CIO_GET_STAT operation. A user-mode process can wait for the next available status block by issuing a **mpselect** entry point with the specified POLLPRI event.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, CIO_START_DONE). The following are the seven possible MPQP status blocks:

CIO_START_DONE Status Block

On a successfully completed CIO_START operation, the status block is filled in as follows:

code	CIO_START_DONE.
option[0]	CIO_OK.
option[1]	Specifies the network ID.
option[2]	Not used.
option[3]	Not used.

On an unsuccessful Start Device (CIO_START) **mpioctl** operation, the status block is filled in as follows:

code	CIO_START_DONE.
option[0]	Can be one of the following:

MP_ADAP_NOT_FUNC	Adapter not functional.
MP_TX_FAILSAFE_TIMEOUT	Transmit command did not complete.
MP_DSR_ON_TIMEOUT	DSR failed to come on.
MP_DSR_ALRDY_ON	DSR already on for a switched line.
MP_X21_RETRIES_EXC	X.21 retries exceeded — call not completed.
MP_X21_TIMEOUT	X.21 timer expired.
MP_X21_CLEAR	Unexpected clear received from the DCE.

option[1]	If the **option[0]** field is set to MP_X21_TIMEOUT, the **option[1]** field contains the specific X.21 timer that expired.
option[2]	Not used.
option[3]	Not used.

CIO_HALT_DONE Status Block

The CIO_HALT ends a session with the MPQP device handler. On a successfully completed Halt Device operation, the status block is filled in as follows:

code	CIO_HALT_DONE.
option[0]	CIO_OK.
option[1]	MP_FORCED_HALT or MP_NORMAL_HALT.
option[2]	MP_NETWORK_FAILURE or MP_HW_FAILURE.

A *forced halt* is a halt that completes successfully as far as the data link control is concerned, but which is terminated forcefully because of either an adapter error or a network error. This is significant for X.21 or other switched networks where customers may be charged if the call does not disconnect properly.

CIO_TX_DONE Status Block

On completion of a Multiprotocol Transmit, the status block is filled in as follows:

code	CIO_TX_DONE.
option[0]	Can be one of the following:

CIO_OK.	
MP_TX_UNDERRUN.	
MP_X21_CLEAR.	
MP_TX_FAILSAFE_TIMEOUT	The transmit command did not complete.
MP_TX_ABORT	Transmit aborted due to CIO_HALT operation.

option[1]	Identifies the **write_id** field supplied by the caller in the write command if TX_ACK was selected.
option[2]	Points to the buffer with transmit data.
option[3]	Not used.

CIO_ASYNC_STATUS Status Block

Asynchronous status notifies the data link control of asynchronous events such as network and adapter failures.

code	CIO_ASYNC_STATUS.
option[0]	Can be one of the following: • MP_X21_CLEAR • MP_RCV_TIMEOUT • MP_DSR_DROPPED • MP_RELOAD_CMPL • MP_RESET_CMPL. **Note:** The MP_RELOAD_C and MPLMP_RESET_CMPL values are for diagnostic use only.
option[1]	Not used.
option[2]	Not used.
option[3]	Not used.

MP_RDY_FOR_MAN_DIAL Status Block

The manual dial switched line is ready for dialing. The start operation is pending the call completion.

code	MP_RDY_FOR_MAN_DIAL
option[0]	CIO_OK
option[1]	Not used
option[2]	Not used
option[3]	Not used.

MP_END_OF_AUTO_RESP Status Block

The MP_STOP_AR **mpioctl** operation has completed. The adapter is in Normal Receive mode. All receive data is routed to the data link control.

code	MP_END_OF_AUTO_RESP
option[0]	CIO_OK
option[1]	Not used
option[2]	Not used
option[3]	Not used.

MP_THRESH_EXC Status Block

A threshold for one of the counters defined in the start profile has reached its threshold.

code	MP_THRESH_EXC.
option[0]	Indicates the threshold that expired. The following values are returned to indicate the threshold that was exceeded: • MP_TOTAL_TX_ERR • MP_TOTAL_RX_ERR • MP_TX_PERCENT • MP_RX_PERCENT.
option[1]	Not used.
option[2]	Not used.
option[3]	Not used.

Execution Environment

The CIO_GET_STAT operation can be called from the process environment only.

Return Values

ENXIO	Indicates the adapter number is out of range.
ENOMEM	Indicates that no **mbufs** or **mbuf** clusters are available.

Implementation Specifics

The CIO_GET_STAT operation functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **mpwrite** entry point, **mpioctl** entry point.

The CIO_START operation, CIO_HALT operation, MP_STOP_AR operation.

CIO_HALT mpioctl Operation (Halt Device MPQP)

Purpose

Ends a session with the Multiprotocol (MPQP) device handler. The CIO_HALT operation terminates the connection to the MPQP link.

Description

The CIO_HALT operation terminates a session with the MPQP device handler. The caller specifies which network ID to halt. The CIO_HALT operation removes the network ID from the network ID table and disconnects the physical link. A CIO_HALT operation must be issued for each CIO_START operation that completed successfully.

Data received for the specified network ID before the CIO_HALT operation is called can be retrieved by the caller by the **mpselect** and **mpread** entry points.

If the CIO_HALT operation terminates abnormally, the status is returned either asynchronously or as part of the CIO_HALT_DONE. Whatever the case, the CIO_GET_STAT is used to get information about the error. When a halt is terminated abnormally (for example, due to network failure), the following occurs:

- The link is terminated.
- The drivers and receivers are disabled for the indicated port.
- The port can no longer transmit or receive data.

There is no recovery procedure required by the caller other than logging the error.

Errors are reported on halt operations because in some switched networks, the user could continue to be charged for connect time if the network does not recognize the halt. This error status permits a network application to be notified about an abnormal link disconnection and then take corrective action, if necessary.

Parameter Block

For the MPQP CIO_HALT operation the *extptr* parameter points to a **session_blk** structure. This structure is defined in the **<sys/comio.h>** file and contains the following fields:

status Specifies the status of the port. This field is set for immediately detectable errors. Possible values for the status filed are:

- CIO_OK
- CIO_NETID_INV.

If the calling process does not wish to sleep while the halt is in progress, the DNDELAY option can be used. In either case, the status of the halt is retrieved using the CIO_GET_STATUS operation and a CIO_HALT_DONE status block is returned. The CIO_HALT_DONE status block should be used as an indication of completion.

netid Contains the network ID the caller wishes to halt. The network ID is placed in the least significant byte of the **netid** field.

Execution Environment

The CIO_HALT operation can be called from the process environment only.

Return Values

The CIO_HALT operation returns common communications return values. In addition, the following MPQP specific errors may be returned:

EBUSY	Indicates that the device is not started or is not in a data transfer state.
ENXIO	Indicates that the adapter number is out of range.
ENOMEM	Indicates that there are no **mbufs** or **mbuf** clusters available.

Implementation Specifics

The CIO_HALT operation functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **mpselect** entry point, **mpread** entry point.

The CIO_START operation, CIO_GET_STATUS operation.

MPQP Status Blocks.

CIO_QUERY mpioctl Operation (Query Statistics)

Purpose

Provides the means to read counter values accumulated by the Multiprotocol (MPQP) device handler.

Description

The CIO_QUERY operation is used by the caller to read the counter values accumulated by the MPQP device handler. The counters are initialized to 0 (zero) by the first **mpopen** operation.

The device-specific information placed in the supplied buffer by the CIO_QUERY operation is:

- DDS header section
- DDS hardware section
- DDS device characteristics
- DDS RAS log section.

The mp_query Parameter Block

For this operation, the *extptr* parameter points to an **mp_query** structure. This structure is defined in the **<sys/mpqp.h>** header file and has the following fields:

qp	Points to the **query_parms** common parameter block. The **query_parms** structure can be found in the **<sys/comio.h>** header file.
buffptr	Specifies the address of a buffer where the returned statistics are to be placed. This field is in the **query_parms** structure.
bufflen	Specifies the length of the buffer. It should be at least 45 words long (unsigned long).This field is in the **query_parms** structure.
mpqp	Points to the **mp_query_parms** structure, which contains the device-specific counters.
clear_counters	When this field equals CIO_CLEAR_CNT, the RAS log counters are cleared upon completion of call.

Statistics Logged for MPQP Ports

The following statistics are logged for each MPQP port.

- Bytes transmitted
- Bytes received
- Frames transmitted
- Frames received
- Receive errors
- Transmission errors
- DMA buffer not large enough or not allocated
- Autoresponse transmission failsafe time out

- Autoresponse received time out
- CTS time out
- CTS dropped during transmit
- DSR time out
- DSR dropped
- DSR on before DTR on a switched line
- X.21 call-progress signal (CPS)
- X.21 unrecognized CPS
- X.21 invalid CPS
- DCE clear during call establishment
- DCE clear during data phase
- X.21 T1 to T5 time outs
- X.21 invalid DCE provided information (DPI).

Execution Environment

The CIO_QUERY operation can be called from the process environment only.

Return Values

ENXIO	Indicates an attempt to use unconfigured device.
EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates an invalid parameter.
EIO	Indicates that an error has occurred.
ENOMEM	Indicates that the operation was unable to allocate the required memory.

Implementation Specifics

The CIO_QUERY operation functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **mpioctl** entry point, **mpopen** entry point.

CIO_START mpioctl Operation (Start Device)

Purpose

Starts a session with the Multiprotocol (MPQP) device handler.

Description

The CIO_START operation registers a network ID in the network ID table and establishes the physical connection with the MPQP device. Once this start operation completes successfully, the port is ready to transmit and receive data.

Note: The CIO_START operation defines the protocol- and configuration-specific attributes of the selected port. All bits that are not defined must be set to 0 (zero).

For the MPQP CIO_START operation, the *extptr* parameter points to a **t_start_dev** structure. This structure contains pointers to the **session_blk** structure.

The **session_blk** structure contains the **netid** and **status** fields. The **t_start_dev** device-dependent information for an MPQP device follows the session block. All of these structures can be found in the **mpqp.h** file.

t_start_dev Fields

The **t_start_dev** structure contains the following fields:

phys_link
Indicates the physical link protocol. The following values are the six supported values.

Physical Link	Type
PL_232D	EIA-232D
PL_422A	EIA-422A
PL_V35	V.35
PL_X21	X.21
PL_SMART_MODEM	Hayes autodial protocol
PL_V25	V.25bis autodial protocol.

Only one type of physical link is valid at a time. Smartmodem and V25bis are both EIA232–D autodial protocols. They imply EIA232–D.

If the **phys_link** field is PL_SMART_MODEM or PL_V25, the **dial_proto** and **dial_flags** fields are applicable. Otherwise, these fields are ignored. If no dial protocol or flags are supplied when PL_SMART_MODEM or PL_V25 is selected, defaults are used. The defaults for the dial phase for a PL_SMART_MODEM link is an asynchronous protocol at 2400 baud with even parity, 7 bits per character, and 1 stop bit. A PL_V25 link has the same defaults.

dial_proto
Identifies the autodial protocol. This is a protocol used for communicating with the modem when sending information such as dial sequence or register settings.

Note: The **dial_proto** field is ignored if the physical link is not an autodial protocol.

Most modems use an asynchronous protocol during the connect phase of call establishment. If no value is set, the default mode is asynchronous.

data_proto Identifies the possible data protocol selections during the data transfer phase. The **data_flags** field has different meanings depending on what protocol is selected. The **data_proto** field accepts the following values:

DATA_PRO_BSC	
DATA_PRO_SDLC_FDX	Receivers enabled during transmit.
DATA_PRO_SDLC_HDX	Receivers disabled during transmit.

modem_flags Establishes modem characteristics. This field accepts the following values:

MF_LEASED	Indicates a leased telephone circuit.
MF_SWITCHED	Indicates a switched telephone circuit.
MF_CALL	Indicates an outgoing call (switched only).
MF_LISTEN	Indicates an incoming call (switched only).
MF_AUTO	Indicates that the call is to be answered or dialed automatically.
MF_MANUAL	Indicates that the operator answers or dials the call manually.
MF_CDSTL_OFF	Indicates that the distribution tape reel (DTR) should be enabled without waiting for ring indicate (RI) (connect data set to line).
MF_CDSTL_ON	Enable DTR after RI occurs. If the DSR goes active prior to RI, DTR is enabled and RI is ignored.
MF_DRS_ON	Enable DRS (date rate selected).
MF_DRS_OFF	Disable DRS (full speed). This is the default.

poll_addr Identifies the address-compare value for a Binary Synchronous Communication (BSC) polling frame or an SDLC frame. If using BSC, a value for the selection address must also be provided or address compare is not enabled. If a frame is received that does not match the poll address (or select address for BSC), the frame is not passed to the system.

select_addr For BSC only, must specify a valid select address.

modem_int_mask
Reserved. This value must be 0 (zero).

baud_rate Specifies the baud rate for transmit and receive clock. This field is used for DTE clocking only (that is, when the modem does not supply the clock). Acceptable baud rates range from 150 baud to a maximum speed of 38400 baud. If this field contains a value that does not match one of the following choices, the next lowest baud rate is used:

- 38400
- 19200
- 9600
- 4800
- 2400
- 2000
- 1200
- 1050
- 600
- 300.

A value of 0 (zero) indicates that the port is to be externally clocked (that is, use modem clocking). The on-board baud rate generator is limited to a speed of 38400. All higher baud rates up to the maximum of 64000 bits must be accomplished with modem clocking. For RS232, the adapter uses BMC clocking and drives a clock signal on the DTE Clock. Most modems provide their own clocking.

If the physical link is set to SMART_MODEM or V.25 bis, the baud rate is the speed of the dial sequence and modem clocking is used for data transfer.

rcv_timeout Indicates the period of time, expressed in 100-msec units (.1 sec), used for setting the receive timer. The MPQP device driver starts the receive timer whenever the CIO_START operation completes and a final transmit occurs.

If a receive occurs that is not a receive final frame, the timer is restarted. The timer is stopped when the receive final occurs. If the timer expires before a receive occurs, an error is reported to the logical link control protocol (LLC). After the CIO_START operation completes, the receive time-out value can be changed by the MP_CHANGE_PARAMS operation. A value of 0 (zero) indicates that a receive timer should not be activated.

Final frames in SDLC are all frames with the poll or final bit set. In BSC, all frames are final frames except ITB frames.

rcv_data_offset
Indicates the offset into the data area of a receive buffer where the MPQP device driver is to begin placing the receive data. This allows the LLC process to force word alignment if required. A minimum value of 6 is used.

dial_data_length
Specifies the length of the dial data. Dial data for Hayes-style dial data can be up to 256 bytes.

Flag Fields for autodial Protocols

Flag fields in the **t_start_dev** structure take different values depending on the type of autodial protocol selected.

Data Flags for the BSC Autodial Protocol

If BSC is selected in the **data_proto** field, either ASCII or EBCDIC character sets can be used. Control characters are stripped automatically on reception. DLEs are automatically inserted and deleted in transparent mode. The values for the control characters are determined by the value of the BA bit. If BSC Address Check mode is selected, values for both poll and select addresses must be supplied. Odd parity is used if ASCII is selected.

The default values are the following:

- EBCDIC
- Do not restart the receive timer
- Do not check addresses
- RTS controlled.

The data flag fields for the BSC autodial protocol are the following:

DATA_FLG_RST_TMR	Reset receive timer.
DATA_FLG_ADDR_CHK	Address compare select. This causes frames to be filtered by the hardware based on address. Only frames with matching addresses are sent to the system.
DATA_FLG_BSC_ASC	ASCII BSC select.
DATA_FLG_C_CARR_ON	Continuous Carrier (RTS always on).
DATA_FLG_C_CARR_OFF	RTS disabled between transmits (default).

Dial Flags for ASC Protocols

If ASC and the parity enable bit is set, the value for parity select is honored. A value of 0 (zero) equals even parity. A value of 1 (one) equals odd parity. If parity enable is set to 0, no parity type is enforced. The following are the acceptable ASC dial flags:

DIAL_FLG_PAR_EN	Enable parity.
DIAL_FLG_PAR_ODD	Odd parity.
DIAL_FLG_STOP_0	0 Stop bits.
DIAL_FLG_STOP_1	1 Stop bit.
DIAL_FLG_STOP_15	1.5 Stop bits.
DIAL_FLG_STOP 2	2 Stop bits.
DIAL_FLG_CHAR_5	5 bits per character.
DIAL_FLG_CHAR_6	6 bits per character
DIAL_FLG_CHAR_7	7 bits per character.
DIAL_FLG_CHAR_8	8 bits per character.
DIAL_FLG_C_CARR_ON	Continuous carrier (RTS always on).
DIAL_FLG_C_CARR_OFF	RTS disabled between transmits (default).
DIAL_FLG_TX_NO_CTS	Transmit without waiting for CTS.
DIAL_FLG_TX_CTS	Wait for CTS to transmit (default).

Data Flags for the SDLC Protocol

For the SDLC protocol, the flag for NRZ or NRZI must match the data-encoding method that is used by the remote DTE. If SDLC Address Check mode is selected, the poll address byte must also be specified. The receive timer is started whenever a final block is transmitted. If RT is set to 1, the receive timer is restarted after expiration. If RT is set to 0, the receive timer is not restarted after expiration. The receive timer value is specified by the 16-bit **rcv_time out** field. The following are the acceptable SDLC data flags:

DATA_FLG_NRZI	NRZI select (default is NRZ).
DATA_FLG_ADDR_CHK	Address compare select.
DATA_FLG_RST_TMR	Restart receive timer.
DIAL_FLG_C_CARR_ON	Continuous carrier (RTS always on).
DIAL_FLG_C_CARR_OFF	RTS disabled between transmits (default).

t_auto_data Fields

The **t_auto_data** structure contains the following fields that specify aspects of the X.21 Call Progress Signal Retry and Logging Data format:

len	Length of autodial to be sent to the modem.
sig[]	Signals to be sent to the modem data in the form of an array of characters.
connect_timer	Time-out value. This value is specified in 0.1 second adapter should wait for call to complete before reporting a connection failure to the DLC. The default is 30 seconds if no value is set.
v25b_tx_timer	Time-out value. This value is specified in 0.1 second of delay after driving DTR and before sending dial data in V.25bis modem protocol. If no value is set, a default value of 0.1 second is used.

t_x21_data Fields

The **t_x21_data** structure contains the following fields that specify aspects of the X.21 Call Progress Signal Retry and Logging Data format:

selection signal length	Contains the length in bytes of the data in the selection signals portion of the buffer. Values from 0 to 256 are valid.
selection signals	The selection signals are allocated 256 bytes each. Items are stored in the International Alphabet 5 (IA5) format.
retry_cnt	Indicates how many attempts at outgoing call establishment must fail before the adapter software returns an error to the driver for the CIO_START operation. Values between 0 and 255 are allowed. This is a 1-byte field.
retry_delay	Contains the number of 100-msec (0.1 sec) intervals to wait between successive call retries. This is a 2-byte field.
cps_group	There are nine characters-per-second (CPS) groups. Each CPS group can generate a driver interrupt after a configurable number of errors are detected. Optionally, this interrupt can cause an X.21 network transaction to notify network error-logging monitors of excessive error rates. The **netlog** bit definitions determine which signals in each group are considered countable.

Retry and Netlog Groups

Specify the **retry** and **netlog** fields for a CPS-particular group. The bits definitions are as follows:

- In the retry field, a 1-bit (ON) indicates that retries are enabled for this signal.
- In logging fields, a set bit indicates that errors of this type should be counted in the cumulative group error statistics. Eventually, these statistics can generate interrupts to the driver.

Call-progress signals are divided into groups of 10, for example, CPS 43 is group 4, signal 3. To indicate retries for CPS 43, the value for signal 3 should be ORed into the retry unsigned short for group 4. Possible values for retry groups are the following:

- CG_SIG_0
- CG_SIG_1
- CG_SIG_2
- CG_SIG_3
- CG_SIG_4
- CG_SIG_5
- CG_SIG_6
- CG_SIG_7
- CG_SIG_8
- CG_SIG_9.

t_err_threshold Fields

The **t_err_threshold** structure describes the format for defining thresholds for transmit and receive errors. Counters track the total number of transmit and receive errors. Individual counters track certain types of errors. Thresholds can be set for individual errors, total errors, or a percentage of transmit and receive errors from all frames received.

When a counter reaches its threshold value, a status block is returned by the driver. The status block indicates the type of error counter that reached its threshold. If multiple thresholds are reached at the same time, the first expired threshold in the list is reported as having expired and its counter is reset to 0 (zero). The user can issue a CIO_QUERY operation call to retrieve the values of all counters.

If no thresholding is desired, the threshold should be set to 0 (zero). A value of 0 indicates that LLC should not be notified of an error at any time. To indicate that the LLC should be notified of every occurrence of an error, the threshold should be set to 1 (one).

The **t_err_threshold** structure contains the following fields:

tx_err_thresh	Specifies the threshold for all transmit errors. Transmit errors include transmit underrun, clear to send (CTS) dropped, CTS time out, and transmit failsafe time out.
rx_err_thresh	Specifies the threshold for all receive errors. Receive errors include overrun errors, break/abort errors, framing/CRC/FCS errors, parity errors, bad frame synchronization, and receive-DMA-buffer-not-allocated errors.
tx_err_percent	Specifies the percentage of transmit errors that must occur before a status block is sent to the LLC.
rx_err_percent	Specifies the percentage of receive errors that must occur before a status block is sent to the LLC.

Execution Environment

The CIO_START operation can be called from the process environment only.

Return Values

ENXIO	Indicates that the adapter number is out of range.
ENOMEM	Indicates that the no **mbufs** or **mbuf** clusters are available.
EBUSY	Indicates that the port state is not opened for a CIO_START operation.
EIO	Indicates that the device handler could not queue command to the adapter.
EFAULT	Indicates that the cross-memory copy service failed.
EINVAL	Indicates that the physical link parameter is invalid for the port.

Implementation Specifics

The CIO_START operation functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **mpioctl** entry point.

The CIO_QUERY operation, MP_CHANGE_PARAMS operation.

MP_CHG_PARMS mpioctl Operation (Change Parameters)

Purpose

Permits the data link control (DLC) to change certain profile parameters after the Multiprotocol (MPQP) device has been started.

Description

The MP_CHG_PARMS operation permits the data link control (DLC) to change certain profile parameters after the MPQP device has been started. The *cmd* parameter in the **mpioctl** entry point is set to MP_CHG_PARMS. This operation can interfere with communications that are in progress. Data transmission should not be active when this operation is issued.

For this operation, the *extptr* parameter points to a **chng_params** structure. This structure has following four fields, which can be changed:

chg_mask	Specifies the mask that indicates which fields are to be changed. The possible choices are: CP_RCV_TMR, CP_POLL_ADDR, and CP_SEL_ADDR. More than one field can be changed with one MP_CHG_PARAMS operation.
rcv_timer	Identifies the time out value used after transmission of final frames when waiting for receive data in .1 second units.
poll_addr	Specifies the poll address. Possible values are SDLC or BSC poll addresses.
sel_addr	Specifies the select address. BSC is the only possible protocol that supports select addresses.

Related Information

The **mpioctl** entry point.

MP_START_AR and MP_STOP_AR mpioctl Operations (Autoresponse)

Purpose

Permits the Multiprotocol (MPQP) port to exit or enter Autoresponse mode.

Description

The MP_START_AR and MP_STOP_AR operations permit the MPQP port to enter and exit Autoresponse mode. When the *cmd* parameter is set to MP_STOP_AR, the device exits from Autoresponse mode. All received data is sent up to the host. The data link control (DLC) receives an end-of-autoresponse status in the **status_block** structure of the CIO_GET_STAT operation.

When the *cmd* parameter is set to MP_START_AR, the port is put into Autoresponse mode. The DLC supplies the address and control bytes for receive compare and transmit in the **t_auto_resp** structure pointed to by the *extptr* parameter. This structure contains the following fields:

rcv_timer	Identifies the time in100-msecs units that the adapter waits after a frame has been transmitted before reporting an error.
tx_rx_addr	The 1-byte address that is used for compare on the receive frames and as the address byte on transmitted frames.
tx_cntl	Specifies the control byte that is used for transmitted frames.
rx_cntl	Identifies the value of control byte on receive frames used for receive compare.

Autoresponse mode is applicable for SDLC protocol only. Autoresponse is used to reduce the amount of system overhead during nonproductive link communications. While DTEs are exchange control information to maintain the link, the adapter can respond to polls from the host without generating any system interrupts.

When in Autoresponse mode, the MPQP adapter compares the receive address and control bytes with the values supplied by the DLC. If a match is found, it generates a response frame with the address and control bytes given in the MP_START_AR operation. When a response frame is transmitted, a timer is started with the value given in the **rcv_timer** field. If the adapter does not receive a frame before the timer expires, an error is detected and Autoresponse mode is exited.

The following five conditions cause the MPQP adapter to exit Autoresponse mode.

- A receive time out occurs.
- A transmit time out occurs.
- A poll/final frame is received that does not compare with the control data given in the autoresponse operation.
- A fatal link error occurs. Fatal errors include data rate select (DSR) dropped and X.21 cleared received.
- A stop autoresponse command is received from the dlc.

If one of these errors occurs, the adapter exits Autoresponse mode and stays in receive mode. Polls received after these errors occur are passed to the DLC.

Execution Environment

The autoresponse operations can be called from the process environment only.

Implementation Specifics

The **mpopen** entry point functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Return Values

ENXIO	Indicates the adapter number is out of range.
ENOMEM	Indicates the no **mbufs** or **mbuf** clusters are available.

Implementation Specifics

The auto-response operations function with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The CIO_GET_STAT operation.

mpmpx Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Provides allocation and deallocation of a channel for the Mulitiprotocol (MPQP) device handler.

Syntax

int mpmpx (*devno*, *chanp*, *channame***)**
dev_t *devno*;
int **chanp*;
char **channame*;
int *openflag*;

Parameters

devno Specifies the major and minor device numbers.

chanp Identifies the channel ID passed as a reference parameter. If the *channame* parameter is NULL, this is the ID of the channel to be deallocated. Otherwise, this parameter is set to the ID of the allocated channel.

channame Points to the remaining path name describing the channel to be allocated. There are four possible values:

Equal to NULL	Deallocates the channel.
Pointer to a NULL string	Allows a normal open sequence of the device on the channel ID generated by the **mpmpx** entry point.
D	Allows the device to be opened in Diagnostic mode on the channel ID generated by the **mpmpx** entry point.
Pointer to W	Allows the MPQP device to be opened in Diagnostic mode with the adapter in Wrap mode on the channel ID generated by the **mpmpx** entry point.

Description

The **mpmpx** entry point provides allocation and deallocation of a channel. This entry point is supported in the same manner as the common **ddmpx** entry point.

Return Values

EINVAL	Indicates an invalid parameter.
ENXIO	Indicates that the device was already open and that the Diagnostic mode open request was denied.
EBUSY	Indicates that the device was already open in Diagnostic mode and that the open request was denied.

Related Information

The **ddmpx** common entry point.

The **mpopen** entry point.

mpopen Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Prepares the Multiprotocol (MPQP) device for transmitting and receiving data.

Syntax

#include <sys/comio.h>
#include <sys/mpqp.h>

int mpopen (*devno*, *devflag*, *chan*, *ext*)
dev_t *devno*;
ulong *devflag*;
int *chan*;
STRUCT kopen_ext **ext*;

Parameters

devno	Specifies the major and minor device numbers.
devflag	Specifies the flag word. For kernel-mode processes, the *devflag* parameter must be set to the **DKERNEL**, flag which specifies that a kernel routine is making the **mpopen** call. In addition, the following flags can be set:

DWRITE	Open for reading and writing.
DREAD	Open for a trace.
DNDELAY	Open without waiting for the operation to complete. If this flag is set, write requests return immediately and read requests return with 0 (zero) length data if no read data is available. The calling process does not sleep. The default is DELAY or blocking mode.
DELAY	This is the default. Wait for the operation to complete before opening.

Note: For user-mode processes, the **DKERNEL** flag must be clear.

chan	Specifies the channel number assigned by the **mpmpx** entry point.
ext	Points to the **kopen_ext** parameter block for kernel-mode processes. Specifies the address to the **mpopen** parameter block for user-mode processes.

Description

The **mpopen** entry point prepares the MPQP device for transmitting and receiving data. This entry point is invoked in response to a **fp_open** kernel service call. The file system in user mode also calls the **mpopen** entry point when an **open** subroutine is issued. The device should be opened for reading and writing data.

Each port on the MPQP adapter must be opened by its own **mpopen** call. Only one open call is allowed for each port. If more than one open call is issued, an error is returned on subsequent **mpopen** calls.

The MPQP device handler only supports one kernel-mode process to open each port on the MPQP adapter. It supports the mpx routines and structures compatible with the communications I/O subsystem, but it is not a true multiplexed device.

The kernel process must provide a **kopen_ext** parameter block. This parameter block is found in **<sys/comio.h>** file.

For a user-mode process, the *ext* parameter points to the **mpopen** structure. This is defined in the **<sys/comio.h>** file. For calls that do not specify a parameter block, the default values are used.

If adapter features such as the read extended **status** field for binary synchronous communication (BSC) message types as well as other types of information about read data are desired, the *ext* parameter must be supplied. This also requires a **readx** or **readx** subroutine. If a system call is used, user data is returned, although status information is not returned. For this reason it is recommended that **readx** subroutines be used.

Note: A CIO_START operation must be issued before the adapter is ready to transmit and receive data. Write commands are not accepted if a CIO_START operation has not been completed successfully.

Execution Environment

The **mpopen** entry point can be called from the process environment only.

Return Value

ENXIO	Indicates that the port intialization failed. This code could also indicate that the registration of the interrupt failed.
ECHRNG	Indicates that the channel number is out of range (too high).
ENOMEM	Indicates that there were no **mbuf** clusters available.
EBUSY	Indicates that the port is in the incorrect state to receive an open. The port may be already opened or not yet configured.

Implementation Specifics

The **mpopen** entry point functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **mpclose** entry point, **mpmpx** entry point.

The **readx** or **readx** subroutine.

The **fp_open** kernel service.

The CIO_START operation.

mpread Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Provides the means for receiving data from the Multiprotocol (MPQP) device.

Syntax

#include <sys/uio.h>

int mpread(*devno*, *uiop*, *chan*, *ext***)**
dev_t *devno*;
struct uio **uiop*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
uiop	Pointer to an **uio** structure that provides variables to control the data transfer operation. The **uio** structure is defined in the **<sys/uio.h>** header file.
chan	Specifies the channel number assigned by the **mpmpx** routine.
ext	Specifies the address of the **read_extension** structure. If the *ext* parameter is NULL, then no parameter block is specified.

Description

Note: Only user-mode processes should use the **mpread** entry point.

The **mpread** entry point provides the means for receiving data from the MPQP device. When a user-mode process user issues a **read** or **readx** subroutine, the kernel calls the **mpread** entry point.

The **DNDELAY** flag, set either at open time or later by an **mpioctl** operation, controls whether **mpread** calls put the caller to sleep pending completion of the call. If a program issues an **mpread** entry point with the **DNDELAY** flag clear (the default), program execution is suspended until the call completes. If the **DNDELAY** flag is set, the call always returns immediately. The user must then issue a poll and CIO_GET_STAT operation to be notified when read data is available.

When user application programs invoke the **mpread** operation through the **read** or **readx** subroutine, the returned length value specifies the number of bytes read. The **status** field in the **read_extension** parameter block should be checked to determine if any errors occurred on the read. One frame is read into each buffer. Therefore, the number of bytes read depends on the size of the frame received.

For a nonkernel process, the device handler copies the data into the buffer specified by the caller. The size of the buffer is limited by the size of the internal buffers on the adapter. If the size of the user buffer exceeds the size of the adapter buffer, the maximum number of bytes on a **mpread** entry point is the size of the internal buffer. For MPQP adapter, the maximum frame size is defined in the **mpqp.h** file.

Data is not always returned on a read operation when an error occurs. In most cases, the error causes an error log to occur. If no data is returned, the buffer pointer is NULL. On errors such as buffer overflow, a kernel-mode process receives the error status and the data.

There are also some cases where network data is returned (usually during a CIO_START operation). Network data is distinguished from normal receive data by the **status** field in the **read_extension** structure. A nonzero status in this field indicates an error or information about the data.

Note: The MPQP device handler uses fixed length buffers for transmitting and receiving data. The **RX_BUF_LEN** field in the **<sys/mpqp.h>** header file defines the maximum buffer size.

The read_extension Parameter Block

For the **mpread** entry points, the *ext* parameter may point to a **read_extension** structure. This structure is found in the **<sys/comio.h>** header file and contains the following field:

status Specifies the status of the port. There are six possible values for the returned status parameter. These status values accompany a data buffer:

CIO_OK	
MP_BUF_OVERFLOW	Indicates receive buffer overflow. For MP_BUF_OVERFLOW, the data that was received before the buffer overflowed is returned with the overflow status.
MP_X21_CPS	Holds an X.21 call progress signal.
MP_X21_DPI	Holds X.21 DCE-provided information (network data).
MP_MODEM_DATA	Contains modem data (for example, an autodial sent by the modem)
MP_AR_DATA_RCVD	Contains data received while in Autoresponse mode.

Execution Environment

The **mpread** entry point can be called from the process environment only.

Return Values

The **mpread** entry point returns the number of bytes read. In addition, this entry point may return one of the following:

ECHRNG	Indicates the channel number was out of range.
ENXIO	Indicates that the port is not in the proper state for a read.
EINTR	Indicates that the sleep was interrupted by a signal.
EINVAL	Indicates the read was called by a kernel process.

Implementation Specifics

The **mpread** entry point functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **mpmpx** entry point, **mpwrite** entry point.

The **read** or **readx** subroutine.

The CIO_START operation, MP_START_AR operation.

The **uio** structure.

mpselect Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Provides the means for determining if specified events have occurred on the Multiprotocol (MPQP) device.

Syntax

#include <sys/devices.h>
#include <sys/comio.h>

int mpselect (*devno*, *events*, *reventp*, *chan***)**
dev_t *devno*;
ushort *events*;
ushort **reventp*;
int *chan*;

Parameters

devno	Specifies the major and minor device numbers.
events	Identifies the events that are to be checked.
reventp	Returned events pointer. This parameter is passed by reference and is used by the **mpselect** entry point to indicate which of the selected events are true at the time of the call.
chan	Specifies the channel number assigned by the **mpmpx** entry point.

Description

Note: Only user-mode processes can use the **mpselect** entry point.

The **mpselect** entry point provides the means for determining if specified events have occurred on the MPQP device. This entry point is supported the same as the common **ddselect** communications entry point.

Execution Environment

The **mpselect** entry point can be called from the process environment only.

Return Values

ENXIO	Indicates an attempt to use an unconfigured device.
EINVAL	Indicates that the select was called from a kernel process.
ECHNG	Indicates the channel number is too large.

Implementation Specifics

The **mpselect** entry point functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **mpmpx** entry point.

The **ddselect** communications entry point.

The **select** system call, **poll** system call.

mpwrite Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Provides the means for transmitting data to the Multiprotocol (MPQP) device.

Syntax

#include <sys/uio.h>
#include <sys/comio.h>
#include <sys/mpqp.h>

int mpwrite (*devno*, *uiop*, *chan*, *ext***)**
dev_t *devno*;
struct uio **uiop*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure that provides variables to control the data transfer operation. The **uio** structure is defined in the **<sys/uio.h>** header file.
chan	Specifies the channel number assigned by the **mpmpx** entry point.
ext	Can specify the address of the **mp_write_extension** parameter block. If the *ext* parameter is NULL, then no parameter block is specified.

Description

The **mpwrite** entry point provides the means for transmitting data to the MPQP device. The kernel calls it when a user-mode process issues a **write** or **writex** subroutine. The **mpwrite** entry point can also be called in response to a **fpwrite** kernel service.

The mpwrite Parameter Block

For the **mpwrite** operation, the *ext* parameter points to the **mp_write_extension** structure. This structure is defined in the **<sys/comio.h>** file. The **mp_write_extension** structure contains the following fields:

status Identifies the status of the port. There are five possible values for the returned **status** field:

CIO_OK	Indicates that the operation was successful.
CIO_TX_FULL	Unable to queue any more transmit requests.
CIO_HARD_FAIL	Hardware failure.
CIO_INV_BFER	Invalid buffer (length equals 0, invalid address).
CIO_NOT_STARTED	Device not yet started.

write_id Contains a user-supplied correlator. The **write_id** field is returned to the caller by the CIO_GET_STAT operation if the CIO_ACK_TX_DONE flag is selected in the asynchronous status block.

For a kernel user, This field is returned to the caller with the **stat_fn** function that was provided at open time.

In addition to the common parameters, the **mp_write_extension** structure contains a field for selecting Transparent mode for binary synchronous communication (BSC). Any nonzero value for this field causes Transparent mode to be selected. Selecting Transparent mode causes the adapter to insert DLEs before all appropriate control characters. Text sent in Transparent mode is unaltered. Transparent mode is normally used for sending binary files.

Note: If an **mp_write_extension** structure is not supplied, Transparent mode can be implemented by the kernel-mode process by imbedding the appropriate DLE sequences in the data buffer.

Execution Environment

The **mpwrite** entry point can be called from the process environment only.

Return Values

EAGAIN Indicates that the number of DMAs has reached the maximum allowed or that the device handler cannot get memory for internal control structures.

Note: The MPQP device handler does not currently support the **tx_fn** function. If a value of EAGAIN is returned by an **mpwrite** entry point, the application is responsible for retrying the write.

ECHRNG Indicates that the channel number is too high.

EINVAL Indicates one of the following:

- The port is not set up properly.
- The MPQP device handler could not set up structures for the write.
- The port is invalid.

ENOMEM Indicates that no **mbuf** structure or clusters are available or the total data length is more than a page.

ENXIO Indicates one of the following:

- The port has not been successfully started.
- An invalid adapter number was passed.
- The specified channel number is illegal.

Implementation Specifics

The **mpwrite** entry point functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **mpread** entry point, **mpopen** entry point.

The CIO_GET_STAT operation.

The **uio** structure.

The **write** or **writex** subroutine.

tokclose Token-Ring Device Handler Entry Point

Purpose

Resets the Token-Ring device handler to a known state and frees system resources.

Syntax

#include <sys/device.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokclose (*devno*, *chan***)**
dev_t *devno*;
int *chan*;

Parameters

devno	Specifies the major and minor device numbers.
chan	Identifies the channel number assigned by the **tokmpx** entry point.

Description

The **tokclose** entry point is called when a user-mode caller issues a **close** subroutine. The **tokclose** entry point can also be invoked in response to a **fp_close** kernel service.

Execution Environment

The **tokclose** entry point can be called from the process environment only.

Return Value

ENXIO	Indicates that an invalid minor number was specified.

Implementation Specifics

The **tokclose** entry point functions with an Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **ddclose** communications entry point.

The **tokopen** entry point, **tokmpx** entry point.

The **fp_close** kernel service.

The **close** subroutine.

tokconfig Token-Ring Device Handler Entry Point

Purpose

Provides functions for initializing, terminating, and querying the vital product data (VPD) of the Token-Ring device handler.

Syntax

#include <sys/device.h>
#include <sys/uio.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokconfig (*devno*, *cmd*, *uiop***)**
dev_t *devno*;
int *cmd*;
struct uio **uiop*;

Parameters

devno	Specifies the major and minor device numbers.
cmd	Identifies the function to be performed by the **tokconfig** routine.
uiop	Points to a **uio** structure that describes the relevant data area for reading or writing.

Description

The **tokconfig** entry point provides functions for initializing, terminating, and querying the vital product data (VPD) of the Token-Ring device handler. The **tokconfig** routine is invoked at device configuration time. The **tokconfig** entry point provides the following three operations:

- **CFG_INIT**

  The CFG_INIT operation initializes the Token-Ring device handler. The Token-Ring device handler registers the entry points in the device switch table.

  The Token-Ring define device structure (DDS) address and length is described in the **uio** structure. The DDS is copied into an internal save area by the device handler.

- **CFG_TERM**

  The CFG_TERM operation terminates the Token-Ring device handler. If there are no outstanding opens, the Token-Ring device handler marks itself terminated and prevents subsequent opens. All dynamically allocated areas are freed. All Token-Ring device handler entry points are removed from the device switch table.

- **CFG_QVPD**

  The CFG_QVPD operation returns the Token-Ring VPD to the caller. The VPD is placed in the area specified by the caller in the **uio** structure.

Execution Environment

The **tokconfig** entry point can be called from the process environment only.

Return Values

Depending on the operation selected, the **tokconfig** entry point returns the following values:

Return Values for the CFG_INIT Operation

ENOMEM	Indicates that the routine was unable to allocate space for the DDS.
EEXIST	Indicates that the device was already initialized.
EINVAL	Indicates the the DDS provided is invalid.
ENXIO	Indicates that the initialization of the Token-Ring device failed.
EFAULT	Indicates that an invalid address was specified.

Return Values for the CFG_TERM Operation

EBUSY	Indicates that there are outstanding opens, unable to terminate.
ENOENT	Indicates that there was no device to terminate.
EACCESS	Indicates that the device was not configured.
EEXIST	Unable to remove the device from the device switch table.

Return Values for the CFG_QVPD Operation

ENOENT	Indicates that there was no device to query the VPD.
EFAULT	Indicates that an invalid address was specified.
EACCESS	Indicates that the Token-Ring device handler is not initialized.

Implementation Specifics

The **tokconfig** entry point functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **uio** structure.

POWERstation and POWERserver Hardware Technical Reference — Options and Devices.

tokioctl Token-Ring Device Handler Entry Point

Purpose

Provides various functions for controlling the Token-Ring device handler.

Syntax

#include <sys/device.h>
#include <sys/devinfo.h>
#include <sys/ioctl.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokioctl (*devno*, *cmd*, *arg*, *devflag*, *chan*, *ext*)
dev_t *devno*;
int *cmd*, *arg*;
ulong *devflag*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
cmd	Specifies the operation to be performed. The possible **tokioctl** operation codes can be found in the **<sys/ioctl.h>**, **<sys/comio.h>**, and **<sys/tokuser.h>** header files.
arg	Specifies the address of the **tokioctl** parameter block.
devflag	Indicates the conditions under which the device was opened.
chan	Specifies the channel number assigned by the **tokmpx** entry point.
ext	This parameter is not used by the Token-Ring device handler.

Description

The **tokioctl** entry point provides various functions for controlling the Token-Ring device handler. There are nine possible **tokioctl** operations:

IOCINFO	I/O Character Information.
CIO_START	Starts the device.
CIO_HALT	Halts the device.
CIO_QUERY	Queries device statistics.
CIO_GET_STAT	Gets device status.
TOK_GRP_ADDR	Sets the group address.
TOK_FUNC_ADDR	Sets functional addresses.
TOK_QVPD	Queries vital product data (VPD).
TOK_RING_INFO	Queries Token-Ring Information.

Execution Environment

The **tokioctl** entry point can be called from the process environment only.

Implementation Specifics

The **tokioctl** entry point functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **tokmpx** entry point.

POWERstation and POWERserver Hardware Technical Reference — Options and Devices.

CIO_GET_STAT tokioctl Operation (Get Status)

Purpose

Gets the current status of theToken-Ring adapter and device handler.

Description

Note: Only user-mode callers can use the CIO_GET_STAT operation.

The CIO_GET_STAT operation returns the current status of the Token-Ring adapter and device handler. For this operation, the *arg* parameter points to the **status_block** structure. This structure is defined in the **<sys/comio.h>** file and takes the following status codes:

- CIO_LOST_DATA
- CIO_NULL_BLK
- CIO_START_DONE
- CIO_HALT_DONE
- CIO_TX_DONE
- CIO_ASYNC_STATUS.

Status Blocks for the Token-Ring Device Handler

Status blocks are used to communicate status and exception information to user-mode processes.

User-mode processes receive a status block whenever they request a CIO_GET_STAT operation. A user-mode process can wait for the next available status block by issuing a **tokselect** entry point with the specified POLLPRI event.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, CIO_START_DONE).

The following status blocks are returned by the Token-Ring device handler:

CIO_START_DONE Status Block

On a successfully completed CIO_START operation, the status block is filled in as follows:

code	CIO_START_DONE.
option[0]	CIO_OK.
option[1]	The low-order 2 bytes are filled in with the **netid** field passed with the CIO_START operation. If a medium access control (MAC) frame session was requested, this field is set to TOK_MAC_FRAME_NETID.
option[2]	The high-order 2 bytes are filled in with the high-order 2 bytes of the network address. The low-order 2 bytes are filled in with the middle 2 bytes of the network address.
option[3]	The high-order 2 bytes are filled in with the low-order 2 bytes of the network address.

If the CIO_START operation is unsuccessful, the status block is filled in as follows:

code	CIO_START_DONE.

option[0]	Can be one of the following: • CIO_TIMEOUT • TOK_ADAP_CONFIG • TOK_ADAP_INIT_PARMS_FAIL • TOK_ADAP_INIT_FAIL • TOK_ADAP_INIT_TIMEOUT • TOK_LOBE_MEDIA_TST_FAIL • TOK_PHYS_INSERT • TOK_ADDR_VERIFY_FAIL • TOK_RING_POLL • TOK_REQ_PARMS.
option[1]	The low-order 2 bytes are filled in with the **netid** field passed with the CIO_START operation. If a MAC frame session was requested, this field is set to TOK_MAC_FRAME_NETID.
option[2]	Adapter Return Code. For each of the device-specific codes returned in **option[0]**, an adapter return code is placed in the low-order 2 bytes of this field. Possible values for the **option[2]** field are the adapter reset, initialization, and open completion codes.
option[3]	Not used.

CIO_HALT_DONE Status Block

On a successfully completed CIO_HALT operation, the status block is filled in as follows:

code	CIO_HALT_DONE.
option[0]	CIO_OK.
option[1]	The low-order 2 bytes are filled in with the **netid** field passed with the CIO_HALT operation. If a MAC frame session was requested, this field is set to TOK_MAC_FRAME_NETID.
option[2]	Not used.
option[3]	Not used.

CIO_TX_DONE Status Block

When a **tokwrite** entry point completes for which transmit acknowledgment has been requested, the following status block is built and returned to the caller.

code	CIO_TX_DONE.
option[0]	CIO_OK or TOK_TX_ERROR.
option[1]	Contains the **write_id** field specified in the **write_extension** structure passed to the **tokwrite** operation.
option[2]	For a kernel-mode process, contains the mbuf pointer that was passed in the **tokwrite** operation.
option[3]	The high-order 2 bytes contain the adapter's transmit command complete code that the adapter returns. The low-order 2 bytes contain the adapter's transmit CSTAT completion code that is returned when a packet is transmitted by the adapter.

CIO_ASYNC_STATUS Status Block

The Token-Ring device handler can return the following types of asynchronous status:

- CIO_HARD_FAIL:
  - TOK_ADAP_CHECK
  - TOK_PIO_FAIL
  - TOK_RCVRY_THRESH
- CIO_NET_RCVRY_ENTER
- CIO_NET_RCVRY_EXIT
- TOK_RING_STATUS

When a CIO_HARD_FAIL status block is returned, the Token-Ring adapter is no longer functional. The user should begin shutting down the Token-Ring device handler.

Hard Failure Status Block Values

Unrecoverable Adapter Check

When an unrecoverable adapter check has occurred, this status block is returned:

code	CIO_ASYNC_STATUS.
option[0]	CIO_HARD_FAIL.
option[1]	TOK_ADAP_CHECK.
option[2]	The adapter return code is in the high-order 2 bytes. The adapter returns three parameters when an adapter check occurs. Parameter 0 (zero) is returned in the low-order 2 bytes.
option[3]	The high-order 2 bytes contain parameter 1 (one). The low-order 2 bytes contain parameter 2.

Unrecoverable PIO Error

When an unrecoverable PIO error has occurred, this status block is returned:

code	CIO_ASYNC_STATUS
option[0]	CIO_HARD_FAIL
option[1]	TOK_PIO_FAIL
option[2]	Not used
option[3]	Not used.

Exceeded Network Recovery Entry Threshold

When the Token-Ring device handler has exceeded the Network Recovery mode entry threshold, this status block is returned:

code	CIO_ASYNC_STATUS
option[0]	CIO_HARD_FAIL
option[1]	TOK_RCVRY_THRESH

option[2]	Not used
option[3]	Not used.

Entered Network Recovery Mode Status Block

When the Token-Ring device handler has entered Network Recovery mode, this status block is returned:

code	CIO_ASYNC_STATUS.
option[0]	CIO_NET_RCVRY_ENTER.
option[1]	Specifies the reason for entering Network Recovery mode. Can be one of these seven options: • TOK_RING_STATUS • TOK_LOBE_WIRE_FAULT • TOK_AUTO_REMOVE • TOK_ADAP_CHECK • TOK_CMD_FAIL • TOK_REMOVE_RECEIVED • TOK_MC_ERROR.
option[2]	Specifies the adapter return code. For an adapter check, the adapter return code is in the high-order 2 bytes. The adapter returns three parameters when an adapter check occurs. The adapter check parameter 0 (zero) is returned in the low-order 2 bytes.
option[3]	For an adapter check, the high-order 2 bytes contain parameter 1 (one). The low-order 2 bytes contain parameter 2.

Exited Network Recovery Mode Status Block

When the Token-Ring device handler has exited Network Recovery mode, the status block is filled in as follows:

code	CIO_ASYNC_STATUS
option[0]	CIO_NET_RCVRY_EXIT
option[1]	Not used
option[2]	Not used
option[3]	Not used.

Ring Beaconing Status Block Values

When the Token-Ring adapter detects a beaconing condition on the ring, it notifies the device handler. The device handler returns the following status block:

code	CIO_ASYNC_STATUS.
option[0]	TOK_RING_STATUS.
option[1]	TOK_RING_BEACONING.
option[2]	Specifies the adapter return code. The low-order 2 bytes contains the ring status.
option[3]	Not used.

Ring Recovered Status Block Values

When the Token-Ring detects that the beaconing condition has ceased, it notifies the device handler. The device handler returns the following status block:

code	CIO_ASYNC_STATUS
option[0]	TOK_RING_STATUS
option[1]	TOK_RING_RECOVERED
option[2]	Not used
option[3]	Not used.

Execution Environment

The CIO_GET_STAT operation can be called from the process environment only.

Return Values

EACCESS	Indicates an illegal call from a kernel-mode user.
EINVAL	Indicates that an invalid parameter was specified.
EFAULT	Indicates that an invalid address was supplied.

Implementation Specifics

The CIO_GET_STAT operation functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The CIO_START operation, CIO_HALT operation.

The **tokioctl** entry point, **tokwrite** entry point.

Network Recovery Mode for the Token-Ring Device Handler.

CIO_HALT tokioctl Operation (Halt Device)

Purpose

Ends a session with the Token-Ring device handler.

Description

The CIO_HALT operation ends a session with the Token-Ring device handler. The caller indicates the network ID that is to be halted. This CIO_HALT operation corresponds to the CIO_START operation that was successfully issued with the specified network ID. A CIO_HALT operation should be issued for each CIO_START operation that was successfully issued.

Data for the specified network ID is no longer received. Data that was received for the specified network ID before the halt is still passed up to a user mode caller by the **tokselect** and **tokread** entry points. Data is passed back to a kernel mode caller by the **rx_fn** routine specified at open time.

For the CIO_HALT operation, the *arg* parameter points to the **session_blk** structure. This structure is defined in the **<sys/comio.h>** file and contains the following fields:

status	Returns one of the following status values: • CIO_OK • CIO_NETID_INV.
netid	Specifies the network ID. The network ID is placed in the least significant byte of the **netid** field. When terminating the MAC Frame session, the **netid** field should be set to TOK_MAC_FRAME_NETID.

Execution Environment

The CIO_HALT operation can be called from the process environment only.

Return Values

EINVAL	Indicates an invalid parameter.
EFAULT	Indicates that an invalid address was specified.
ENOMSG	Indicates that an error occurred.

Implementation Specifics

The CIO_HALT operation functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The CIO_START operation, the CIO_GET_STAT operation.

The **tokselect** entry point, **tokread** entry point, **tokioctl** entry point.

CIO_QUERY tokioctl Operation (Query Statistics)

Purpose

Allows the caller to read the counter values accumulated by the Token-Ring device handler.

Description

The CIO_QUERY operation is used by the caller to read the counter values accumulated by the Token-Ring device handler. The first call to the **tokopen** entry point initializes the counters to 0 (zero).

For the CIO_QUERY operation, the *arg* parameter points to the **query_parms** structure. This structure is defined in the **<sys/comio.h>** header file and contains the following fields:

status	Indicates the status of the port. This field may be CIO_OK or CIO_INV_CMD.
buffptr	Specifies the address of a buffer where the returned statistics are to be placed.
bufflen	Specifies the length of the buffer.
clearall	When this value equals CIO_QUERY_CLEAR, the counters are cleared upon completion of call. The CIO_QUERY_CLEAR label can be found in the **<sys/comio.h>** header file.

The counters placed in the supplied buffer by the CIO_QUERY operation are the counters declared in the **tok_query_stats_t** structure defined in the **<sys/tokuser.h>** header file.

Execution Environment

The CIO_QUERY operation can be called from the process environment only.

Return Values

EFAULT	Indicates than an invalid address was specified.
EINVAL	Indicates an invalid parameter.

Implementation Specifics

The CIO_QUERY operation functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **tokioctl** entry point, **tokopen** entry point.

POWERstation and POWERserver Hardware Technical Reference — Options and Devices.

CIO_START (Start Device tokioctl Operation)

Purpose

Initiates a session with the Token-Ring device.

Description

The CIO_START operation initiates a session with the Token-Ring device handler. If the start is the first on the port, the device handler initializes and opens the Token-Ring adapter. For each successful CIO_START call issued, there should be a corresponding CIO_HALT operation issued.

After the CIO_START operation has successfully completed, the adapter is ready to transmit and receive data. The caller is free to issue any valid Token-Ring operation. Once started, the adapter receives packets that match the Token-Ring adapter's (hardware) address (or the address specified in the DDS) and broadcast. No group or functional address is specified when the adapter is brought up.

The caller notifies the device handler of the network ID that it wishes to use. The network ID corresponds to the destination service access point (DSAP) in the Token-Ring packet. The caller can issue multiple CIO_START operations. For each adapter the Token-Ring device handler can handle from 0 (zero) to the number of network IDs specified by the TOK_MAX_NETIDS label. This label is defined in the **<sys/tokuser.h>** header file.

The session_blk Parameter Block

For the CIO_START operation, the *arg* parameter points to the **session_blk** structure. This structure is defined in the **<sys/comio.h>** header file and contains the following fields:

status Indicates the status of the CIO_START. Possible returned **status** values are:

CIO_OK

CIO_NETID_FULL

CIO_NETID_DUP.

netid Specifies the network ID the caller will use on the network. The Network ID is placed in the least significant byte of the **netid** field. To request a MAC frame session, the **netid** field should be set to the TOK_MAC_FRAME_NETID label. This value has a unique identifier in the most significant byte of the **netid** field. There can be only one MAC frame session per adapter.

Note: The AIX Token-Ring device handler does not allow the caller to specify itself as the wild card network ID.

Execution Environment

The CIO_START operation can be called from the process environment only.

Return Values

EINVAL	Indicates an invalid parameter.
ENETDOWN	Indicates an unrecoverable hardware error.
ENOMSG	Indicates an error.
ENOSPC	Indicates the network ID table is full.
EADDRINUSE	Indicates the network ID is in use.

Implementation Specifics

The CIO_START operation functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **tokioctl** entry point.

The CIO_HALT operation, CIO_GET_STAT operation.

IOCINFO tokioctl Operation (Describe Device)

Purpose

Returns a structure that describes the Token-Ring device.

Description

The IOCINFO operation returns a structure that describes the Token-Ring device. For this operation, the *arg* parameter points to the **devinfo** structure. This structure is defined in the **<sys/devinfo.h>** file and contains the following fields:

devtype	Identifies the device type. The Token-Ring device type is DD_NET_DH. This value is defined in the **<sys/devinfo.h>** file.
devsubtype	Identifies the device subtype. The Token-Ring device subtype is DD_TR. This value is defined in the **<sys/devinfo.h>** file.
speed	Specifies the capabilities of the Token-Ring device. This is equal to TOK_4M when the Token-Ring device is configured with a data rate of 4 Mbps. The capabilities are TOK_16M when the Token-Ring device is configured with a data rate of 16 Mbps. The TOK_4M and TOK_16M labels are defined in the **<sys/tokuser.h>** file.
broad_wrap	Specifies whether the wrapping of broadcast packets is supported by the device.
rdto	Specifies the configured receive data transfer offset (RDTO) value.
haddr	Specifies the 6-byte hardware address of the Token-Ring adapter card.
net_addr	Specifies the 6-byte Network address currently used by the Token-Ring device handler.

The parameter block is filled in with the appropriate values upon return.

Execution Environment

The IOCINFO operation can be called from the process environment only.

Return Values

EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates an invalid parameter.
ENXIO	Indicates that an invalid minor number was specified.

Implementation Specifics

The IOCINFO operation functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **tokioctl** entry point.

TOK_FUNC_ADDR tokioctl Operation (Set Functional Address)

Purpose

Specifies a functional address to be used on the Token-Ring device.

Description

The TOK_FUNC_ADDR operation allows the caller to specify a functional address to be used on the Token-Ring network. A successful CIO_START operation must be issued before a TOK_FUNC_ADDR operation can be issued. The parameter block for the functional address is the **tok_func_addr_t** structure defined in the **<sys/tokuser.h>** header file.

The **tok_func_addr_t** structure has four fields:

status	Returns the one of the following status conditions: • CIO_OK • CIO_NOT_STARTED • CIO_NETID_INV • CIO_TIMEOUT • CIO_INV_CMD.
netid	Specifies the network ID associated with this functional address. The network ID must have previously been successfully started by the CIO_START operation. There can only be one functional address specified per network ID.
opcode	When set to TOK_ADD, the functional address is added to the possible functional addresses the Token-Ring adapter accepts packets for. When set to TOK_DEL, the functional address is removed from the possible functional addresses to accept packets for. The TOK_ADD and TOK_DEL are defined in the **<sys/tokuser.h>** header file.
func_addr	Specifies the functional address. The most significant bit and the two least significant bits cannot be set. They are ignored by the Token-Ring adapter. **Note:** On the Token-Ring network, a group address is a 6-byte address. The most significant 2 bytes are automatically compared to a 0xC000 by the Token-Ring adapter.

Execution Environment

The TOK_GRP_ADDR operation can be called from the process environment only.

Return Values

ENOMSG	Indicates that an error occurred.
EFAULT	Indicates that an invalid address was specified.
ENETDOWN	Indicates an unrecoverable hardware error.
EINVAL	Indicates an invalid parameter.
ENOCONNECT	Indicates that the device has not been started.

Implementation Specifics

The TOK_GRP_ADDR operation functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **tokioctl** entry point.

The CIO_START operation.

Token-Ring Status Blocks.

POWERstation and POWERserver Hardware Technical Reference — Options and Devices.

TOK_GRP_ADDR tokioctl Operation (Set Group Address)

Purpose

Sets the active group address for the Token-Ring adapter.

Description

The TOK_GRP_ADDR operation sets the active group address for the Token-Ring adapter. Only one group address can be specified at a time for a Token-Ring adapter. For this operation, the *arg* parameter points to the **tok_group_addr_t** structure. This structure is defined in the **<sys/tokuser.h>** header file and contains the following fields:

status Returns on of the following possible status values:

- CIO_OK
- CIO_NOT_STARTED
- CIO_TIMEOUT
- CIO_INV_CMD
- TOK_NO_GROUP.

opcode When set to TOK_ADD, the group address specified is added to the possible address, the Token-Ring device accepts packets for. When set to TOK_DEL, the group address is removed from the possible receive packet addresses. The TOK_ADD and TOK_DEL values are defined in the **<sys/tokuser.h>** header file.

group_addr Specifies the group address. The Token-Ring adapter ignores the most significant bit of this field.

Note: On the Token-Ring network a group address is a 6-byte address. The most significant 2 bytes are automatically compared to a 0xC000 by the Token-Ring adapter.

Execution Environment

The TOK_GRP_ADDR operation can be called from the process environment only.

Return Values

ENOMSG	Indicates than an error occurred.
EFAULT	Indicates that an invalid address was specified.
ENETDOWN	Indicates an unrecoverable hardware error.
EINVAL	Indicates an invalid parameter.
ENOCONNECT	Indicates that the device has not been started.

Implementation Specifics

The TOK_GRP_ADDR operation functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **tokioctl** entry point.

Token-Ring Status Blocks, Token-Ring Operation Results.

TOK_QVPD tokioctl Operation (Query Vital Product Data)

Purpose

Returns the vital product data (VPD) for the Token-Ring adapter.

Description

The TOK_QVPD operation returns VPD about the Token-Ring device. For this operation, the *arg* parameter points to the **tok_vpd_t** block for the query vital product data (VPD). This structure is defined in the **<sys/tokuser.h>** header file and contains the following fields:

status Returns one of the following status conditions:

- **TOK_VPD_VALID**
- **TOK_VPD_NOT_READ**
- **TOK_VPD_INVALID.**

l_vpd Specifies the length of the *vpd* parameter.

vpd[TOK_VPD_LENGTH]
Contains the VPD upon return.

Execution Environment

The TOK_QVPD operation can be called from the process environment only.

Return Values

EINVAL	Indicates an invalid parameter.
EFAULT	Indicates that an invalid address was specified.
ENXIO	Indicates that an invalid minor number was specified.

Implementation Specifics

The TOK_QVPD operation functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **tokioctl** entry point.

TOK_RING_INFO tokioctl Operation (Query Token-Ring)

Purpose

Reads information about the Token-Ring device.

Description

The TOK_RING_INFO operation is used by the caller to read information about the Token-Ring device. For this operation, the *arg* parameter points to the **tok_q_ring_info_t** structure. This structure is defined in the **<sys/tokuser.h>** header file and contains the following fields:

status	Indicates the status condition that occurred. Possible status values are: • CIO_OK • CIO_NOT_STARTED • TOK_NO_RING_INFO.
p_info	Points to the buffer where the **tok_ring_info_t** structure is to be copied. The **tok_ring_info_t** structure is defined in the **<sys/tokuser.h>** file.
l_buf	Specifies the length of the buffer for the returned Ring Information structure.

Execution Environment

The TOK_RING_INFO operation can be called from the process environment only.

Return Values

EINVAL	Indicates an invalid parameter.
EFAULT	Indicates that an invalid address was specified.
ENOMSG	Indicates that an error occurred.
ENOCONNECT	Indicates that the device has not been started.

Implementation Specifics

The TOK_RING_INFO operation functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **tokioctl** entry point.

Token-Ring Operation Results, List of Common Status/Exception Codes, Token Ring Status Block Values.

POWERstation and POWERserver Hardware Technical Reference — Options and Devices.

tokmpx Token-Ring Device Handler Entry Point

Purpose

Provides allocation and deallocation of a channel for the Token-Ring device handler.

Syntax

#include <sys/device.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokmpx (*devno*, *chanp*, *channame*)
dev_t *devno*;
int **chanp*;
char **channame*;

Parameters

devno	Specifies the major and minor device numbers.
chanp	Specifies the channel ID passed as a reference parameter. If the *channame* parameter is NULL, this is the ID of the channel to be deallocated. Otherwise, this parameter is set to the ID of the allocated channel.
channame	Points to the remaining path name describing the channel to be allocated. The *channame* parameter accepts the following values:

Equal to NULL	Deallocates the channel.
Pointer to a NULL string	Allows a normal open sequence of the Token-Ring device on the channel ID generated by the **tokmpx** entry point.
Pointer to D	Allows the Token-Ring device to be opened in Diagnostic mode on the channel ID generated by the **tokmpx** entry point.
Pointer to W	Allows the Token-Ring device to be opened in Diagnostic mode with the adapter in Wrap mode on the channel ID generated by the **tokmpx** entry point.

Description

The **tokmpx** entry point is not called directly by a user of the Token-Ring device handler. The kernel calls the **tokmpx** entry point in response to an open or close request.

If the Token-Ring device has been successfully opened, any Diagnostic mode open requests fail. If the device has been successfully opened in Diagnostic mode, all subsequent open requests fail.

Execution Environment

The **tokmpx** entry point can be called from the process environment only.

Return Values

EACCES	Indicates that the device was already open and that the Diagnostic mode open request was denied.
EBUSY	Indicates that the device was already open in Diagnostic mode and that the open request was denied.
ENOMSG	Indicates that an error occurred.
ENXIO	Indicates an invalid minor number was specified.
ENOSPC	Indicates that the maximum number of opens has been exceeded.

Implementation Specifics

The **tokmpx** entry point functions with an Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **ddopen** entry point, **ddmpx** entry point, **ddclose** entry point, **tokclose** entry point, **tokopen** entry point.

Data Transmission for the Token-Ring Device Handler, Data Reception for the Token-Ring Device Handler.

tokopen Token-Ring Device Handler Entry Point

Purpose

Initializes the Token-Ring device handler and allocates the required system resources.

Include Files

#include <sys/device.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

Kernel-mode Syntax

int tokopen (*devno*, *devflag*, *chan*, *arg*)
dev_t *devno*;
ulong *devflag*;
int *chan*;
struct kopen_ext **arg*;

User-mode Syntax

int tokopen (*devno*, *devflag*, *chan*, *arg*)
dev_t *devno*;
ulong *devflag*;
int *chan*;
int *arg*;

Parameters

devno	Specifies the major and minor device numbers.
devflag	Specifies the flag word with the following definitions:

DKERNEL	This flag is set to indicate a kernel-mode processes. For user-mode processes, this flag must be clear.
DNDELAY	If this flag is set, the device handler performs nonblocking reads and writes for this channel. Otherwise, blocking reads and writes are performed for this channel.

chan	Specifies the channel number assigned by the **tokmpx** entry point.
arg	For kernel-mode processes, points to a **kopen_ext** structure. For user-mode processes, this field is not used.

Description

The **tokopen** entry point is called when a user-mode caller issues an **open**, **openx**, or **creat** subroutine. The **tokopen** routine can also be invoked in response to a **fp_opendev** kernel service. The device is opened for reading and writing of data.

Note: After the **tokopen** operation has successfully completed, the caller must issue a CIO_START operation before any data can be transmitted or received from the Token-Ring device handler.

Execution Environment

The **tokopen** entry point can be called from the process environment only.

Return Values

ENXIO	Indicates that an invalid minor number was specified.
EINVAL	Indicates that an invalid parameter was specified.
ENOMEM	Indicates that the device handler was unable to allocate the required memory.

Implementation Specifics

The **tokopen** entry point functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **ddopen** communications entry point.

The CIO_START operation.

The **open**, **openx**, or **creat** subroutine.

Data Transmission for the Token-Ring Device Handler, Data Reception for the Token-Ring Device Handler.

tokread Token-Ring Device Handler Entry Point

Purpose

Provides the means for receiving data from the Token-Ring device handler.

Syntax

#include <sys/device.h>
#include <sys/uio.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokread (*devno*, *uiop*, *chan*, *arg*)
dev_t *devno*;
struct uio **uiop*;
int *chan*;
read_extension **arg*;

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure. For a calling user-mode process, the **uio** structure specifies the location and length of the caller's data area in which to transfer information. The kernel fills in the **uio** structure for the user.
chan	Specifies the channel number assigned by the **tokmpx** entry point.
arg	Can be NULL or else point to the **read_extension** structure. This structure is defined in the **<sys/comio.h>** header file

Description

Note: Only user-mode callers should use the **tokread** entry point.

The **tokread** entry point provides the means for receiving data from the Token-Ring device handler. When a user-mode caller issues a **read, readx, readv**, or **readvx** subroutine, the kernel calls the **tokread** entry point.

For this operation, the *arg* parameter may point to the **read_extension** structure. This structure is defined in the **<sys/comio.h>** header file and contains the following fields:

status	Contains additional inforamtion about the completion of the **tokread** entry point. Possible values for this field are: • CIO_OK • CIO_BUF_OVFLW.
netid	Not used.
sessid	Not used.

Execution Environment

The **tokread** entry point can be called from the process environment only.

Return Values

EACCESS	Indicates an illegal call from a kernel-mode user.
ENXIO	Indicates that an invalid minor number was specified.
EINTR	Indicates that a system call was interrupted.
EMSGSIZE	Indicates that the data was too large to fit into the receive buffer and that no *arg* parameter was supplied to provide an alternate means of reporting this error with a status of CIO_BUF_OVFLW.
EFAULT	Indicates that an invalid address was supplied.
ENOCONNECT	Indicates that the device has not been started.

Implementation Specifics

The **tokread** entry point functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **tokwrite** entry point, **tokmpx** entry point.

The **read, readx, readv**, or **readvx** subroutine.

List of Common Communications Status/Exception Codes.

tokselect Token-Ring Device Handler Entry Point

Purpose

Determines if a specified event has occurred on the Token-Ring device.

Syntax

#include <sys/device.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokselect (*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 conditions to be checked are denoted by the bitwise OR of one or more of the following:

POLLIN	Check if receive data is available.
POLLOUT	Check if transmit available.
POLLPRI	Check if status is available.
POLLSYNC	Check if asychronous notification is available.

reventp	Points to the result of condition checks. A bitwise OR of the following conditions is returned:

POLLIN	Receive data is available.
POLLOUT	Transmit available.
POLLPRI	Status is available.

chan	Specifies the channel number assigned by the **tokmpx** entry point.

Description

Note: Only user-mode callers should call this entry point.

The **tokselect** entry point is called when the **select** or **poll** subroutine is used to determine if a specified event has occurred on the Token-Ring device.

When the Token-Ring device handler is in a state in which the event can never be satisfied (for example, an adapter failure), then the **tokselect** entry point sets the returned events flags to 1 (one) for the event that cannot be satisfied. This prevents the **select** or **poll** subroutines from waiting indefinitely.

Execution Environment

The **tokselect** entry point can only be called from the process environment.

Return Values

ENXIO	Indicates that an invalid minor number was specified.
EACCES	Indicates an invalid call from a kernel process.

Implementation Specifics

The **tokselect** entry point functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **select** subroutine, **poll** subroutine.

Select/Poll Logic for Write, Read, and Exception-Handling Routines.

tokwrite Token-Ring Device Handler Entry Point

Purpose

Provides the means for transmitting and receiving data to and from the Token-Ring device handler.

Syntax

#include <sys/device.h>
#include <sys/uio.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokwrite (*devno*, *uiop*, *chan*, *arg*)
dev_t *devno*;
struct uio **uiop*;
int *chan*;
struct write_extension **arg*;

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure specifying the location and length of the caller's data.
chan	Specifies the channel number assigned by the **tokmpx** entry point.
arg	Points to a **write_extension** structure. If the *arg* parameter is NULL, then default values are assumed.

Description

The **tokwrite** entry point provides the means for transmitting data to the Token-Ring device handler. The kernel calls it when a user-mode caller issues a **write, writex, writev**, or **writevx** subroutine.

For a user-mode process, the kernel fills in the **uio** structure with the appropriate values. A kernel-mode process must fill in the **uio** structure as described by the **ddwrite** communications entry point.

For the **tokwrite** entry point, the *arg* parameter may point to a **write_extension** structure. This structure is defined in the **<sys/comio.h>** header file and contains the following fields:

status Indicates the status condition that occurred. Possible values for the returned **status** field are:

- CIO_OK
- CIO_TX_FULL
- CIO_NOT_STARTED
- CIO_NET_RCVRY_MODE.

flag May consist of a bitwise OR of the following:

CIO_NOFREE_MBUF
Requests that the PDH not free the **mbuf** structure after transmission is complete. The default is bit clear (free the buffer). For a user-mode process, the PDH always frees the **mbuf** structure.

CIO_ACK_TX_DONE
Requests that, when done with this operation, the PDH acknowledges completion by building a CIO_TX_DONE status block. In addition, requests the PDH either call the kernel status function or (for a user-mode process) place the status block in the status/exception queue. The default is bit clear (do not acknowledge transmit completion).

write_id For a user-mode caller, the **write_id** field is returned to the caller by the CIO_GET_STAT operation (if the CIO_ACK_TX_DONE option is selected). For a kernel-mode caller, the **write_id** field is returned to the caller by the **stat_fn** function that was provided at open time.

Execution Environment

The **tokwrite** entry point can be called from the process environment only.

Return Values

ENXIO	Indicates that an invalid minor number was specified.
ENETDOWN	Indicates that the network is down. The device is unable to process the write.
ENETUNREACH	The device is in Network Recovery mode and is unable to process the entry point.
ENOCONNECT	Indicates that the device has not been started.
EAGAIN	Indicates that the transmit queue is full.
EINVAL	Indicates that an invalid parameter was specified.
ENOMEM	Indicates that the device handler was unable to allocate the required memory.
EINTR	Indicates that a system call was interrupted.
EFAULT	Indicates that an invalid address was supplied.

Implementation Specifics

The **tokwrite** entry point functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **tokmpx** entry point, **tokopen** entry point.

The CIO_START operation.

The **uio** structure.

The **write, writex, writev**, or **writevx** subroutine.

Network Recovery Mode for the Token-Ring Device Handler.

x25sclose X.25 Device Handler Entry Point

Purpose

Closes an X.25 device handler channel.

Syntax

int x25sclose (*devno*, *chan*, *ext*)
dev_t *devno*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
chan	Identifies the channel number assigned by the **x25smpx** entry point.
ext	Not used by the **x25sclose** entry point.

Description

The **x25sclose** entry point closes an X.25 device handler channel. For each channel opened by the **x25sopen** entry point, there must be a corresponding **x25sclose** entry point. When the X.25 device handler receives an **x25sclose** entry point, the device handler frees all internal data areas associated with the corresponding **x25sopen** entry point. In addition, any receive data for the indicated channel is purged.

Note: The **x25sclose** entry point does not free the channel itself. The channel is freed by the **x25smpx** entry point, which the kernel calls immediately after the **x25sclose** entry point.

If the channel being closed is the only open channel for the minor device, the X.25 device handler does the following as well:

- Frees the interrupt level.
- Resets all static data to its initial state.

Before issuing the **x25sclose** entry point, the caller should issue a call to the CIO_HALT operation for each successful CIO_START operation. If the user does not call the CIO_HALT operation (for example, the call was invoked by the kernel after a user process ended abnormally), the X.25 device handler performs the CIO_HALT operation on all open sessions on the channel before continuing with the **x25sclose** function. The close purges all data waiting on the channel. No special clear data can be sent and any clear confirm data is lost.

Note: If the user does not call a CIO_HALT, it is possible data could be lost on the channel's open sessions.

Execution Environment

An **x25sclose** entry point can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. The kernel sets the **errno** global variable to one of the following values:

EINTR	Indicates that the close call was interrupted.
ENXIO	Indicates that the channel was not valid.

Implementation Specifics

The **x25sclose** entry point functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25smpx** entry point, **x25sopen** entry point.

The CIO_START operation, CIO_HALT operation.

x25sioctl X.25 Device Handler Entry Point

Purpose

Provides various functions for controlling the X.25 device.

Syntax

int x25sioctl (*devno*, *cmd*, *arg*, *devflag*, *chan*, *ext***)**
dev_t *devno*;
int *cmd*, *arg*;
ulong *devflag*;
int *chan*, *ext*;

Parameters

devno	Specifies the major and minor device numbers.
cmd	Specifies which of the available **x25sioctl** operations is to be performed.
arg	Identifies the address of the **x25sioctl** parameter block. The meaning of this field depends on the value of the *cmd* parameter.
devflag	Indicates how the device was opened and whether the caller is a user- or kernel-mode process. This parameter accepts the following flags:

DKERNEL	A kernel-mode process called the entry point. This flag is clear if a user-mode process called the entry point.
DREAD	Open for reading. This is the default for the X.25 handler regardless of whether this flag is set.
DWRITE	Open for writing. This is the default for the X.25 handler regardless of whether this flag is set.
DAPPEND	Open for appending. The X.25 handler ignores this flag.
DNDELAY	If this flag is set, the X.25 device handler performs nonblocking reads and writes. Otherwise, blocking reads and writes are performed.

chan	Identifies the channel number assigned by the **x25smpx** entry point.
ext	The extended system call parameter. The meaning of this field depends on the value of the *cmd* parameter.

Description

The **x25sioctl** X.25 device handler entry point provides various functions for controlling the X.25 device. The following are valid operations for the X.25 device:

IOCINFO	Returns a structure that describes the device.
CIO_START	Starts a session and registers a network ID.
CIO_HALT	Halts a session.
CIO_QUERY	Returns the current RAS counter values.
CIO_GET_STAT	Return the next status block.

In addition to the above standard operations, the **x25sioctl** operation supports the following X.25 specific commands:

X25_REJECT	Rejects a call.
X25_QUERY_SESSION	Queries a session.
X25_ADD_ROUTER_ID	Adds a router ID.
X25_DELETE_ROUTER_ID	Deletes a router ID.
X25_QUERY_ROUTER_ID	Queries router ID.
X25_LINK_CONNECT	Connects a link.
X25_LINK_DISCONNECT	Disconnects a link.
X25_LINK_STATUS	Queries the status of a link.
X25_LOCAL_BUSY	Enables or disables receiving of data packets on a port.
X25_COUNTER_GET.	Gets a counter for asynchronous notification.
X25_COUNTER_WAIT	Waits for the contents of counters to change.
X25_COUNTER_READ	Reads the value of a counter.
X25_COUNTER_REMOVE	Removes a counter from the system.
X25_DIAG_IO_WRITE	Writes to an I/O register on the adapter.
X25_DIAG_IO_READ	Reads from an I/O register on the adapter.
X25_DIAG_MEM_WRITE	Writes to memory on the adapter from a user's buffer.
X25_DIAG_MEM_READ	Reads memory from the adapter into a user's buffer.
X25_DIAG_TASK	Downloads the diagnostics task onto the card.

Execution Environment

The **x25sioctl** entry point can be called from the process environment only.

Implementation Specifics

The **x25sioctl** entry point functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25smpx** entry point.

CIO_DNLD x25sioctl (Download Task) Operation

Purpose

Downloads tasks to the kernel.

Description

Note: The CIO_DNLD operation can be called only by a user-mode process. The **DKERNEL** flag must be clear to run this operation.

The CIO_DNLD operation downloads tasks to the kernel. This routine is used to pass microcode between the configuration program and the device driver. Each call to this routine completely replaces any previous version of microcode stored in the device driver.

Notes:

1. The Download Task operation does not download the microcode to the card. It transfers the microcode into kernel memory so that the microcode is available when needed.
2. If the microcode for RCM, X.25, or diagnostics is not available, the code pointer should be set to NULL and the code length set to 0 (zero).

For the CIO_DNLD operation, the *arg* parameter points to a **x25_task** structure. This structure contains the following fields:

x25_code	Points to X.25 code.
rcm_code	Points to real-time microcode (RCM) code.
diagnositic_code	Points to diagnostic code.
x25length	Specifies the length of the X.25 code.
rcm_length	Specifies the length of the RCM code.
diagnostic_length	Specifies the length of the diagnostic code.

Execution Environment

The CIO_DNLD operation can be called from the process environment only.

Return Value

A return code of –1 indicates an unsuccessful operation. The kernel sets the **errno** global variable to the following value:

EFAULT	Indicates that an invalid address was specified.

Implementation Specifics

The CIO_DNLD operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The X.25 ioctl Operations.

CIO_GET_STAT x25sioctl Operation (Get Status)

Purpose

Returns the next status block.

Restrictions

Note: Only a user-mode process can call the CIO_GET_STAT operation.

Description

The CIO_GET_STAT operation reads the next status block available on the channel. This operation does not block. To call the CIO_GET_STAT operation, the **DKERNEL** flag must be clear, indicating a user-mode process. If a new status block is available for a kernel-mode process, the **stat_fn** kernel procedure defined by the **x25sopen** entry point is called.

For the CIO_GET_STAT operation, the *arg* parameter returns a pointer to a **status_block** structure as defined in the **<sys/comio.h>** header file.

Status Blocks for the X.25 Device Handler

Status blocks are provided to user- and kernel-mode processes differently. Kernel-mode processes receive a status block when they are called using the **stat_fn** kernel procedure. This procedure is indicated when a channel is opened by the **x25sopen** entry point.

User-mode processes receive a status block whenever they issue a CIO_GET_STAT operation. In addition, a user-mode process can wait for the next available status block by issuing an **x25sselect** entry point with the DPOLLPRI event specified.

Status blocks can be solicited to indicate a completion of a previous command (such as a CIO_START operation). Status blocks can also be unsolicited to indicate some asynchronous event.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, CIO_START_DONE). The following is a list of the five types of status blocks:

CIO_START_DONE Status Block

This block is provided by the X.25 device handler when the CIO_START operation is complete:

option[0]	Specifies the status or exception code. This option is set to.CIO_OK if the start is successful. Otherwise, it is set to one of the status returns defined by the CIO_START operation.
option[1]	Specifies the session ID and the network ID. The 2 high-order bytes contain the session ID and the 2 low-order bytes contain the network ID.
option[2]	Valid only for a session of type SESSION_SVC_OUT. For a user-mode process, this option contains a pointer to the first byte of the process's data **buffer.** For kernel-mode process, this option contains the **mbuf** structure passed with the CIO_START operation. The **mbuf** structure describes a buffer that represents either a Call Connected or a Clear Indication received in response to a Call Request. This option is NULL for other session types.
option[3]	Not used.

CIO_HALT_DONE Status Block

This block is provided by the X.25 device handler when the CIO_HALT operation is complete:

option[0] Specifies the status or exception code. This option is set to CIO_OK if the start is successful. Otherwise, it is set to one of the status returns defined by the CIO_HALT operation.

option[1] Specifies the session ID and the network ID. The 2 high-order bytes contain the session ID, and the 2 low–order bytes contain the network ID.

option[2] Valid only if the CIO_HALT was used to issue a Clear Request. For a user-mode process, this option contains a pointer to the first byte of the process's data buffer.

For kernel-mode process, this option contains the **mbuf** structure passed with the CIO_HALT operation. The buffer described by the **mbuf** structure represents a Clear Confirm received in response to the Clear Request. If there was a clear collision, the buffer represents a Clear Indication that should be treated as an acknowledgment of the Clear Request as no Clear Confirm will be subsequently received.

This option is NULL for other CIO_HALT operation types.

option[3] Not used.

CIO_TX_DONE Status Block

This block is provided by the X.25 device handler when a transmit request for which an acknowledgment was requested is complete:

option[0] Specifies the status or exception code. This option is set to CIO_OK if the start is successful. Otherwise, it is set to one of the status returns defined by the **x25swrite** entry point.

option[1] Specifies the write ID in the **write_extension** structure. This structure is passed by the **x25swrite** entry point.

option[2] Points to the first byte of a user-mode process data buffer or to the **mbuf** structure for a kernel-mode process. The **mbuf** structure is passed with the **x25swrite** entry point.

option[3] Specifies the session ID in the 2 high-order bytes and the network ID in the 2 low–order bytes.

CIO_NULL_BLK Status Block

This status block is returned whenever a status block is requested but none are available:

option[0]	Not used
option[1]	Not used
option[2]	Not used
option[3]	Not used.

X25_REJECT_DONE Status Block

This status block indicates that the CIO_REJECT operation is complete:

option[0]	Specifies the status or exception code. This option is set to CIO_OK if the start is successful. Otherwise, it is set to one of the status returns defined by the CIO_REJECT operation.
option[1]	Specifies the session ID in the 2 high-order bytes and the network ID in the 2 low-order bytes.
option[2]	Identifies the call ID of the incoming call that is being rejected.
option[3]	Not used.

Execution Environment

The CIO_GET_STAT operation can be called from the process environment only.

Return Value

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to the following value:

EFAULT	Indicates that an invalid address was specified.

Implementation Specifics

The CIO_GET_STAT operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sselect** entry point, **x25swrite** entry point, **x25sopen** entry point.

The CIO_START operation, CIO_HALT operation, X25_REJECT operation.

List of Common Status/Exception Codes.

The X.25 mbuf Structure.

CIO_HALT x25sioctl Operation (Halt Session)

Purpose

Ends a session with the X.25 device handler.

Description

The CIO_HALT operation ends a session with the X.25 device handler. A session identified by a particular **session_id** field can be terminated in any of the following five ways:

- With Clear Request on an SVC
- With Clear Confirm on an SVC
- With request for deallocation of a PVC
- With request to stop listening for incoming calls
- By turning off the monitoring facilities on the card.

If the CIO_HALT operation is the last for this port, appropriate termination (such as automatic disconnection, if configured) is done.

The CIO_HALT operation returns immediately to the caller before the halt completes. If the return does not indicate an error, the X.25 device handler builds a CIO_HALT_DONE status block on completion of the operation. For kernel mode processes the status block is passed to the associated status function, specified by the **x25sopen** entry point. For user-mode processes, the block is placed in the associated status and exception queue.

Parameter Block

The parameter block for the CIO_HALT operation is the **x25_halt_data** structure. This structure contains the following fields:

sb	Indicates that the **session_blk** structure defined in the **<sys/comio.h>** file. The **status** field in this structure has meaning when returned only if the return code is EIO.
session_id	Identifies the ID of the session to be halted.

If the CIO_HALT operation is issued to send a Clear Request packet on a session of type SESSION_SVC_OUT or SESSION_SVC_IN, then the CIO_HALT operation *ext* parameter is used. If used, the *ext* parameter points to a buffer containing the data required for the clear request packet This data is in the form described in the **mbuf** structure.

For a kernel-mode process, the data passed in the *ext* parameter is an **mbuf** pointer. Only the calling process can free the **mbuf** data returned in the CIO_HALT_DONE status block. The **mbuf** data returned by this status block is not the same as the data passed down.

For a user-mode process, the data passed in the *ext* parameter is a pointer to a buffer of the same format in user space. If the pointer is NULL, then the clear request is sent with default cause-and-diagnostic codes (0,0), but with no facilities or user data. When the CIO_HALT_DONE status block is received, the buffer is filled with the contents of the clear confirm packet.

Execution Environment

The CIO_HALT operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. The kernel sets the **errno** global variable to one of the following values:

EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates invalid values in the *ext* parameter buffer.
EIO	Indicates that an error has occurred. The **status** field in the **status_block** structure indicates one of the following five common exception codes

- CIO_HARD_FAIL
- CIO_NOMBUF
- CIO_TIMEOUT
- CIO_LOST_DATA
- CIO_NOT_STARTED.

In addition, the following three X.25-specific codes may be returned;

X25_PROTOCOL	Indicates that a protocol error occurred.
X25_NO_LINK	Indicates that the link is not connected.
X25_BAD_PKT_TYPE	Indicates that the packet type passed in the *ext* parameter is not valid. For session types SESSION_SVC_OUT or SESSION_SVC_IN, the packet type should be either PKT_CLEAR_REQ or PKT_CLEAR_CONFIRM.

Implementation Specifics

The CIO_HALT operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point, **x25sopen** entry point.

The CIO_START operation.

The **mbuf** structure.

Sessions with the X.25 Device Handler, X.25 Status Blocks.

CIO_QUERY x25sioctl Operation (Query Device)

Purpose

Returns device statistics and device-dependent information.

Description

Note: The counters and profile information can only be cleared by a system user.

The CIO_QUERY operation returns the **ras_log** field of the define device structure (DDS).

The query_params Parameter Block

For the CIO_QUERY operation the *arg* parameter returns a pointer to the **query_params** structure. The **query_params** structure contains the following fields:

status If the return code is EIO, this field contains the returned value.

bufptr Points to an **x25_query_data** structure. This structure contains the following fields:

- **cc** Contains a **cio_stats** structure as defined in the **<sys/comio.h>** file.
- **ds** Contains an **x25_stats** structure identifying X.25-specific statistics. This structure is found in the **<sys/x25user.h>** file.

buflen Specifies the length of the buffer.

clearall Clears the statistics when set to CIO_QUERY_CLEAR. Any other setting leaves the statistics unchanged.

x25_stats Structure

The **x25_stats** structure identifies X.25-specific statics. Information in this structure includes the **transmit_profile** field. This field provides a profile of the transmission packet sizes in use on a port and can be used for configuration of adapter buffers. The **transmit_profile** field contains a count of the number of packets sent since the device was last configured. The size of these packets must be in the range specified.

Index	**Size**
0	Packet size not known
1	Reserved
2	Reserved
3	Reserved
4	0 to 15
5	16 to 31
6	32 to 63
7	64 to 127

8	128 to 255
9	256 to 511
10	512 to 1023
11	1024 to 2047
12	2048 to 4095
>12	Reserved.

Execution Environment

The CIO_QUERY operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation and the kernel sets the **errno** global variable to one of the following values:

EFAULT Indicates that an invalid address was specified.

EIO Indicates that an error has occurred. The **arg–>status** field contains one of the following common exception codes:

- CIO_BAD_MICROCODE
- CIO_HARD_FAIL
- CIO_NOT_STARTED
- CIO_TIMEOUT
- CIO_LOST_DATA.

EMSGSIZE Indicates that the statistical data was greater then the length of the buffer specified by the **buflen** field. The data in the buffer is truncated.

ENOBUFS Indicates that no buffers are available.

ENXIO Indicates that the device had not been completely configured.

Implementation Specifics

The CIO_QUERY operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

X.25 **ioctl** Operations.

CIO_START x25sioctl Operation (Start Session)

Purpose

Starts an X.25 device handler session.

Description

The CIO_START operation starts an X.25 session. Only one X.25 session is associated with a CIO_START operation. An X.25 session can be initiated by any of the following:

- A call request on a switched virtual circuit (SVC).
- A call accepted on an SVC in response to an incoming call received on some other (listening) session.
- A request for allocation of a permanent virtual circuit (PVC).
- A request to listen for incoming calls satisfying a named specification in the routing table.

If the CIO_START operation is the first one issued for a port, the operation also does the appropriate initialization (for example, downloading the microcode).

The CIO_START operation returns immediately to the caller, before the command completes. If the operation completes successfully, the X.25 device handler builds a CIO_START_DONE status block on completion. For kernel-mode processes, the status block is passed to the associated status function specified at **x25sopen** time. For user-mode processes, the block is placed in the associated status/exception queue.

If the immediate return indicates an error, there is no need to halt the operation. However, if the status block indicates an error, the calling process must issue a halt. An X.25 CIO_HALT operation can be called before a CIO_START_DONE status block is received. In this case, it is undefined whether the session generates a CIO_START_DONE status block.

Note: Read or write operations should not be performed until the CIO_START_DONE status block is received.

Parameter Block

For the CIO_START operation, the *arg* parameter points to an **x25_start_data** structure as defined in the **<sys/comio.h>** file. This structure contains the following seven fields:

sb Defines a **session_blk** structure as described in **<sys/comio.h>** file. This structure contains the following fields:

netid Identifies the network ID. This field can be set by the caller to a correlator that is returned with any data received on this session.

status Identifies return values. This field is meaningful only if the return code is EIO.

session_name Specifies an ASCII name supplied by the caller for RAS purposes. This field is null-terminated if less than 16 characters.

session_id Is a unique ID for this session returned by the X.25 device handler. The caller must use this ID to identify the session on all subsequent calls.

session_type Specifies the type of session required.

The X.25 device handler permits a process to start a session of type SESSION_SVC_IN only if its UID is the same as that of the process that owns the session of type SESSION_SVC_LISTEN that received the incoming call.

If the session type is SESSION_SVC_OUT or SESSION_SVC_IN, then the CIO_START operation *ext* parameter is used. If used, the *ext* parameter points to the data required for the Call Request and Call Accepted packets issued by an out or in session. This data is in the form described in the **mbuf** structure (found in the **<sys/x25user.h>** header file). For a kernel-mode process the data is a **mbuf** pointer. For user-mode, the data is a pointer to a buffer in user space of the same format.

For a SESSION_SVC_OUT session, the **option[2]** field of the status block points to the packet that completed the CIO_START operation. This is either a PKT_CALL_CONNECTED or PKT_CLEAR_INDICATION packet.

session_protocol
Specifies the protocol for this session. This field is set by the caller and is valid only for a SESSION_SVC_OUT or SESSION_SVC_IN session. The protocol for SESSION_PVC is set in the configuration.

The **session_protocol** field accepts the following six values:

PROTOCOL_ELLC
Reserved.

PROTOCOL_QLLC_1980
Selects SNA 1980 cause-and-diagnostic codes instead of CCITT.

PROTOCOL_QLLC_1984
Selects SNA 1984 cause-and-diagnostic codes instead of CCITT.

PROTOCOL_TCPIP
No specific action.

PROTOCOL_YBTS
Yellow Book Transport Service

For this protocol, the X.25 device handler does not handle X.25 packet sequences on behalf of the user. Instead, incoming packets with the M bit set are passed to the user without waiting for the sequence to complete.

PROTOCOL_ISO8208
No specific action. This value is used whenever no other specific protocol is wanted.

counter_id Specifies the counter to increment for any incoming data on this session. This field is set by the calling process. This field set to –1 indicates that counters are not used on this session.

Note: Counter functions are available only to user-mode processes.

session_type_data
Contains additional data set by the caller. The data returned in this field depends on the value of the **session_type** field. The following are the three possible data types:

listen_name Identifies the nickname of an entry (or collection of entries) in the router table. This must be set by the caller with the CIO_START operation when the **session_type** field is set to SESSION_SVC_LISTEN.

call_id Contains the incoming call ID supplied to a listening session by the device handler with an incoming call from remote data terminal equipment (DTE). This value must be set by the caller with the CIO_START operation when the **session_type** field is SESSION_SVC_IN.

logical_channel
Specifies the logical channel number of the PVC to be acquired. This field must be set by the caller with the CIO_START operation when the **session_type** field is set to SESSION_PVC.

Note: When the session type is SESSION_SVC_OUT, no additional data is required.

Execution Environment

The CIO_START operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. The kernel sets the **errno** global variable to one of the following values:

EFAULT Indicates that an invalid address was specified.

EIO Indicates that an error has occurred. The error is returned in the **sb.status** field of the CIOSTART parameter block and takes any one of the following four common exception codes:

- CIO_BAD_MICROCODE
- CIO_HARD_FAIL
- CIO_NOMBUF
- CIO_TIMEOUT.

In addition, the **sb.status** field may take any of the following twelve X.25-specific codes;

X25_NO_LINK Could not connect to the link.

X25_NOT_PVC The channel is not defined as a PVC.

X25_PVC_USED
The PVC is in use by another application.

X25_TOO_MANY_VCS
Too many virtual circuits have been opened.

	X25_PROTOCOL	A protocol error occurred. For example, a SESSION_SVC_IN session was cleared by the remote data terminal equipment (DTE) before it could be accepted. The clear packet can be read using the **x25sread** operation before issuing the halt.
	X25_AUTH_LISTEN	The UID in the router table entry that corresponds to the **listen_name** field does not match the calling UID.
	X25_INV_CTR	The counter specified in the **x25_start_data** field does not exist.
	X25_NAME_USED	The **listen_name** field specified on an SESSION_SVC_LISTEN session is in use by another application.
	X25_NO_NAME	The **listen_name** field specified on an SESSION_SVC_LISTEN session is not in the router table.
	X25_CLEAR	The session has been cleared.
	X25_BAD_CALL_ID	The **call_id** field specified on a SESSION_SVC_IN session is invalid.
	X25_BAD_PKT_TYPE	The packet type passed by the *ext* parameter is not valid.

EINVAL Indicates that any of the following three errors may have occurred:

- The **session_type** field is not valid. This field must be set to PKT_CALL_REQ for a SESSION_SVC_OUT session or to PKT_CALL_ACCEPT for a SESSION_SVC_IN session.
- The **session_protocol** field is not valid.
- The *chan* parameter was not opened in the correct mode. For a SESSION_MONITOR session, the channel must be opened in M mode. For sessions of type SESSION_SVC_IN, SESSION_SVC_OUT, SESSION_SVC_LISTEN, the channel must be opened without a mode.

EINTR Indicates that a signal was received during the call.

ENOBUFS Indicates that there are no spare buffers in the pool.

EBUSY Indicates that the number of starts for this device was exceeded. This occurs with a monitor device that can only support one start.

ENXIO Indicates that the device was not completely configured. Initial configuration must be completed before any starts can be issued.

Implementation Specifics

The CIO_START operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sopen** entry point, **x25sread** entry point, **x25sioctl** entry point.

The CIO_HALT operation.

X.25 **ioctl** Operations, X.25 Status Blocks.

IOCINFO x25sioctl Operation (Identify Device)

Purpose

Returns I/O character information for a X.25 device.

Description

The IOCINFO operation returns I/O character information for an X.25 device. The parameter block for this operation is defined in the **<sys/definfo.h>** header file by the **devinfo** structure. This structure contains the following fields:

devtype	Identifies the type of device. This is set to the DD_X25 value (defined as the ASCII character x).
flags	Undefined for X.25 devices.
devsubtype	Undefined for X.25 devices.

In addition to the previous members, the **devinfo.h** file also contains an **x25** structure (found in the **<sys/x25user.h>** header file). This structure defines the X.25 device and contains the following eight members:

support_level	Identifies a support level of 1980 or 1984.
nua	Contains a null-terminated ASCII string that represents the network-user address.
subscription_facilities_supported	Contains device-dependent information.
network_id	Specifies the identification code for the network. The range and default value for this code is defined by the device configuration.
max_tx_packet_size	Specifies the maximum size of a transmitted data packet. This packet is encoded in the manner of the ISO 8208 definition.
max_rx_packet_size	Specifies the maximum size of a received data packet. This packet is encoded in the manner of the ISO 8208 definition.
default_svc_tx_packet_size	Specifies the default transmit packet size for an switched virtual circuit (SVC). This packet is encoded in the manner of the ISO 8208 definition.
default_svc_rx_packet_size	Specifies the default received packet size for an SVC. This packet is encoded in the manner of the ISO 8208 definition.

Permanent Virtual Circuit (PVC) Packets

PVC packet sizes are configured on a per-PVC basis. To determine the packet size on a PVC you can use either of the following operations:

- The CIO_START operation followed by a X25_QUERY_SESSION operation.
- The CIO_QUERY operation.

Execution Environment

The IOCINFO operation can be called from the process environment only.

Return Value

A return code of –1 indicates an unsuccessful operation. The kernel sets the **errno** global variable to the following value:

EFAULT	Indicates that an invalid address.

Implementation Specifics

The IOCINFO operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The CIO_START operation, X25_QUERY_SESSION operation, CIO_QUERY operation.

Data Transmission for the X.25 Device Handler, Data Reception for the X.25 Device Handler.

X25_ADD_ROUTER_ID x25sioctl Operation (Add Router ID)

Purpose

Registers a new routing name and routing specification.

Description

Note: Only a process that has opened the router special file can call the X.25 X25_ADD_ROUTER_ID operation.

The X25_ADD_ROUTER_ID operation registers a new route name and routing specification in the Router Table. For this operation, the *arg* parameter points to a **x25_router_add** structure. This structure contains the following fields:

router_id	Specifies the unique identifier for the entry. A priority of 1 is high, while 3 indicates a low priority.
listen_name	Specifies the nickname identifier for the entry. The nickname need not be unique.
priority	Identifies the integer priority to attach to the routing request.
action	Specifies the action to be taken if the name is not being listened to. This field takes the following values:

0	Forwards the incoming call so that it can match other listening specifications.
1	Rejects the incoming call with cause 0 (zero), diagnostic 0 (zero).

uid	Identifies the user ID allowed to receive these incoming calls. This field can be the user ID number. A value of –1 indicates that any user ID can receive the calls. Any attempt by a user with insufficient authority to listen on a name is rejected with the EACCES return value.
call_user_data	Contains the call user data to match with an incoming call. The last character can be an * (asterisk). The format of this data is a string of hexadecimal characters and an optional * (asterisk), for example, `C3*`. The call user data is null terminated if it is less than the maximum length.

Additionally, the **x25_router_add** structure contains the following address fields:

- **called_subaddress[20]**
- **calling_address[20]**
- **extended_calling_address[41]**
- **extended_called_address[41]**.

These addresses are set to match with an incoming call. The last character of an address can be an * (asterisk). The addresses are null-terminated if less than the maximum length.

Execution Environment

The X25_ADD_ROUTER_ID operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EFAULT	Indicates that an invalid address was specified.
EACESS	Indicates that the ioctl was issued on an channel that was not opened in Router mode.
EINVAL	Indicates one of the following occurred: • The specified router ID already exists. (Router IDs must be unique.) • The **action** field passed was neither 0 (zero) or 1 (one).
ENOMEM	Indicates that the operation ran out of memory.

Implementation Specifics

The X25_ADD_ROUTER_ID operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

X.25 **ioctl** Operations.

X25_COUNTER_GET x25sioctl Operation (Get Counter)

Purpose

Gets a counter for asynchronous notification.

Description

Note: Only user-mode processes can use counter operations.

The X25_COUNTER_GET operation uses the *arg* parameter to return a counter ID. The ID can be used to wait and test for incoming X.25 data.

Execution Environment

The X25_COUNTER_GET operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EFAULT	Indicates that an invalid address was specified.
ENOSPC	Indicates that there are no counters available to allocate.

Implementation Specifics

The X25_COUNTER_GET operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The X25_COUNTER_WAIT operation, X25_COUNTER_READ operation, X25_COUNTER_REMOVE operation.

Using Counters to Correlate Messages.

X25_COUNTER_READ x25sioctl Operation (Read Counter)

Purpose

Reads the value of a counter.

Description

Note: Only user-mode processes can use counter operations.

The X25_COUNTER_READ operation reads the value of a counter. For this operation, the *arg* parameter points to the **x25_counter_info** structure. This structure contains the following fields:

counter_id	Identifies a counter to read.
counter_value	Holds the current value of the counter on return of the Read Counter operation.

Execution Environment

The X25_COUNTER_READ operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates that the counter ID does not exist.

Implementation Specifics

The X25_COUNTER_READ operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The X25_COUNTER_GET operation, X25_COUNTER_WAIT operation, X25_COUNTER_REMOVE operation.

The **x25sioctl** entry point.

X.25 **ioctl** Operations.

X25_COUNTER_REMOVE x25sioctl Operation (Remove Counter)

Purpose

Removes a counter from the system.

Description

Note: Only user-mode processes can use counter operations.

The X25_COUNTER_REMOVE operation removes the specified counter from the system. For this operation, the *arg* parameter indicates what ID is to be removed. An error code is returned if there is outstanding data to be read associated with this counter.

Execution Environment

The X25_COUNTER_REMOVE operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EBUSY	Indicates that one of the following errors occurred: • There are some packets still waiting to read. • The counter is being waited on by another process.
EACCES	Indicates that the application did not get the counter. The counter is not deleted.
EINVAL	Indicates that the counter ID specified does not exist.

Implementation Specifics

The X25_COUNTER_REMOVE operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The X25_COUNTER_GET operation, X25_COUNTER_WAIT operation, X25_COUNTER_READ operation.

The **x25sioctl** entry point.

X.25 **ioctl** Operations.

X25_COUNTER_WAIT x25sioctl Operation (Wait Counter)

Purpose

Waits for the contents of counters to change.

Description

Note: Only user-mode processes can use counter operations.

The X25_COUNTER_WAIT operation waits for the contents of a counter to change. The process that called this operation is suspended until the value of one of its counters exceeds the value specified by the **counter_value** field.

For the X25_COUNTER_WAIT operation, the *arg* parameter points to the **x25_counter_list** structure. This structure contains the following fields:

counter_num	Identifies the number of elements in the counter array.
counter_array	Specifies an array of the following:

flags	Indicates if the counter information was successfully matched. If successful, the top bit of the flags field is set on return of the Wait Counter operation.
counter_id	Identifies the counter to wait on.
counter_value	Specifies the value the counter must exceed in order for the counters to match successfully.

Execution Environment

The X25_COUNTER_WAIT operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EFAULT	Indicates that an invalid address was specified.
ENOMEM	Indicates that the operation ran out of memory.
EIDRM	Indicates that the counter has been removed.
EINVAL	Indicates that one or more of the counters in the list does not exist.

Implementation Specifics

The X25_COUNTER_WAIT operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

X.25 **ioctl** Operations.

The X25_COUNTER_GET operation, X25_COUNTER_READ operation, X25_COUNTER_REMOVE operation.

X25_DELETE_ROUTER_ID x25sioctl Operation (Delete Router ID)

Purpose

Removes a routing name.

Description

Note: Only a process that has opened the router special file can call the X.25 X25_DELETE_ROUTER_ID operation.

The X25_DELETE_ROUTER_ID operation removes a routing name from the Router Table. For this operation, the *arg* parameter points to the **x25_router_del** structure. This structure contains the following fields:

router_id	Specifies the unique ID for the entry.
override	Indicates how listening is handled. If set to 0 (zero), the routing entry is not deleted if any process is listening for it. If set to a nonzero value, outstanding listens are canceled. No notification is given to the listening applications if the outstanding listens are canceled.

Execution Environment

The X25_DELETE_ROUTER_ID operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one the following values:

EFAULT	Indicates that an invalid address was specified.
EACESS	Indicates that the **ioctl** operation was issued on an channel that was not opened in Router mode.
EBUSY	Indicates that the router ID was being listened to and the override option was not set.
EINVAL	Indicates that the router ID cannot be deleted because it does not exist.

Implementation Specifics

The X25_DELETE_ROUTER_ID operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

X.25 **ioctl** Operations.

X25_DIAG_IO_READ x25sioctl Operation (Read Register)

Purpose

Reads from an I/O register on the IBM X.25 Interface Co-Processor/2.

Description

Note: Only a process that has opened the device for diagnostics can issue this call.

The X25_DIAG_IO_READ operation is used to read from an I/O register on the IBM X.25 Interface Co-Processor/2. Both direct and indirect registers can be read, the card's pointer register is adjusted by this operation.

For this operation, the *arg* parameter returns a pointer to an **x25_diag_io** structure. The value this operation reads is placed in the **value** field of the **x25_diag_io** structure.

Execution Environment

The X25_DIAG_IO_READ operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

ENXIO	The operation attempted to read a card that was not configured.
EACCES	The device was not opened in diagnostic mode.

Implementation Specifics

The X25_DIAG_IO_READ operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The X25_DIAG_IO_WRITE operation, X25_DIAG_MEM_WRITE operation, X25_DIAG_MEM_READ operation, X25_DIAG_TASK operation.

X.25 **ioctl** Operations.

X25_DIAG_IO_WRITE x25sioctl Operation (Write to Register)

Purpose

Writes to an I/O register on the IBM X.25 Interface Co-Processor/2.

Description

Note: Only a process that has opened the device for diagnostics can call this process.

The X25_DIAG_IO_WRITE operation writes to an I/O register on the IBM X.25 Interface Co-Processor/2. Both direct and indirect registers can be written to as the card's pointer register is adjusted by this operation.

Execution Environment

The X25_DIAG_IO_WRITE operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** variable to one of the following values:

ENXIO	The operation attempted to read a card that was not configured.
EACCESS	The channel was not opened in diagnoistic mode.

Implementation Specifics

The X25_DIAG_IO_WRITE operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The X25_DIAG_IO_READ operation, X25_DIAG_MEM_WRITE operation, X25_DIAG_MEM_READ operation, X25_DIAG_TASK operation.

X.25 **ioctl** Operations.

X25_DIAG_MEM_READ x25sioctl Operation (Read Memory)

Purpose

Reads memory from the IBM X.25 Interface Co-Processor/2 into a user's buffer.

Description

Note: Only a process that has opened the device for diagnostics can call this process.

The X25_DIAG_MEM_READ operation reads memory from the IBM X.25 Interface Co-Processor/2 into a user's buffer. For this operation, the *arg* parameter points to a **x25_diag_mem** structure. This structure provides the following:

- Page and offset of card memory to start from
- Number of bytes to read
- Pointer to a buffer into which the data is read.

The read operation can cover more than one page of the card's memory.

Execution Environment

The X25_DIAG_MEM_READ operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** variable to one of the following values:

ENXIO	The operation attempted to read a card that was not configured.
EACCESS	The channel was not opened in diagnostic mode.

Implementation Specifics

The X25_DIAG_MEM_READ operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The X25_DIAG_IO_WRITE operation, X25_DIAG_IO_READ operation, X25_DIAG_MEM_WRITE operation, X25_DIAG_TASK operation.

The **x25sioctl** entry point.

X.25 **ioctl** Operations.

X25_DIAG_MEM_WRITE x25sioctl Operation (Write Memory)

Purpose

Writes memory to the IBM X.25 Interface Co-Processor/2 from a user's buffer.

Description

Note: Only a process that has opened the device for diagnostics can issue this call.

The X25_DIAG_MEM_WRITE operation writes memory to the IBM X.25 Interface Co-Processor/2 from a user's buffer. For this operation, the *arg* parameter points to a **x25_diag_mem** structure. This parameter provides the following:

- Page and offset of card memory to start from
- Number of bytes to write
- Pointer to the user's buffer containg the data to write.

The write can cover more than one page of the card's memory.

Execution Environment

The X25_DIAG_MEM_WRITE operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** variable to one of the following values:

ENXIO	The operation attempted to read a card that was not configured.
EACCESS	The channel was not opened in diagnostic mode.

Implementation Specifics

The X25_DIAG_MEM_WRITE operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The X25_DIAG_IO_WRITE operation, X25_DIAG_IO_READ operation, X25_DIAG_MEM_READ operation, X25_DIAG_TASK operation.

The **x25sioctl** entry point.

X.25 **ioctl** Operations.

X25_DIAG_RESET x25sioctl Operation (Reset Adapter)

Purpose

Resets the IBM X.25 Interface Co-Processor/2.

Description

Note: Only a process that has opened the device for diagnostics can call this process.

The X25_DIAG_RESET operation completely resets the IBM X.25 Interface Co-Processor/2.

Execution Environment

The X25_DIAG_RESET operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** variable to one of the following values:

ENXIO	The operation attempted to read a card that was not configured.
EINVAL	The channel was not opened in diagnoistic mode.

Implementation Specifics

The X25_DIAG_RESET operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The X25_DIAG_IO_WRITE operation, X25_DIAG_IO_READ operation, X25_DIAG_MEM_WRITE operation, X25_DIAG_MEM_READ operation, X25_DIAG_TASK operation.

The **x25sioctl** entry point.

X.25 **ioctl** Operations.

X25_DIAG_TASK x25sioctl Operation (Download Diagnostics)

Purpose

Provides the means to download the diagnostics task onto the card.

Description

The X25_DIAG_TASK operation provides the means to download the diagnostics task onto IBM X.25 Interface Co-Processor/2. The task microcode must have been previously downloaded to the device handler using the CIO_DNLD operation.

For the X25_DIAG_TASK operation, the *arg* parameter points to a **x25_diag_addr** structure that is used to return the load page and offset.

Execution Environment

The X25_DIAG_TASK operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** variable to one of the following values:

ENXIO	The operation attempted to read a card that was not configured.
ENACCES	The channel was not opened in diagnostic mode. You must have appropriate authority open a channel in diagnostic mode.
EINVAL	The microcode was not available to download.

Implementation Specifics

The X25_DIAG_TASK operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The X25_DIAG_IO_WRITE operation, X25_DIAG_IO_READ operation, X25_DIAG_MEM_WRITE operation, X25_DIAG_MEM_READ operation.

X.25 **ioctl** Operations.

X25_LINK_CONNECT x25sioctl Operation (Connect Link)

Purpose

Connects the link to the data circuit-terminating equipment (DCE).

Description

Note: Only a process that has opened the router special file can issue the X25_LINK_CONNECT operation.

The X25_LINK_CONNECT operation connects the X.25 link to the network. The connection is made using the automatic calling unit (ACU), if required. If the link is already connected, no action is taken.

For the X25_LINK_CONNECT operation, the *arg* parameter points to the **x25_connect_data** structure. This structure contains only a **status** field. This field has meaning only when the return code is EIO.

Execution Environment

The X25_LINK_CONNECT operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EIO Indicates that an I/O error occurred. The **status** field in the **x25_connect_data** structure contains one of the following values:

- CIO_BAD_MICROCODE
- CIO_HARD_FAIL
- CIO_TIMEOUT.

EACCES Indicates that the calling application does not have NET_CONFIG authority.

ENOBUFS Indicates that no buffers are available.

Implementation Specifics

The X25_LINK_CONNECT operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The X25_LINK_DISCONNECT operation, X25_LINK_STATUS operation.

X.25 **ioctl** Operations.

X25_LINK_DISCONNECT x25sioctl Operation (Disconnect Link)

Purpose

Disconnects the link to the DCE.

Description

Note: This command is restricted to user programs that have NET_CONFIG permission.

The X25_LINK_DISCONNECT operation disconnects the X.25 link from the network. If the link is already disconnected, no action is taken. If there are virtual calls in progress on the link, disconnection takes place only if the *override* parameter is nonzero.

The X25_LINK_DISCONNECT operation returns synchronously. The X25_LINK_STATUS operation is used to determine if the disconnect operation is complete.

For the X25_LINK_DISCONNECT operation, the *arg* parameter points to an **x25_disconnect_data** structure. This structure contains the following fields:

status	Holds values supplied by the ioctl operation if there is an EIO error.
override	Specifies how disconnection occurs. If this parameter is 0 (zero), the disconnection takes place only if there are no virtual calls in progress. Otherwise, the disconnection is forced. This disconnects the link layer *only*, not the physical layer.

Execution Environment

The X25_LINK_DISCONNECT operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EBUSY Indicates that there are active circuits on the link.

EIO Unable to disconnect due to an I/O error. The **status** field in the **x25_disconnect_data** structure contains one of the following common exception codes:

- CIO_TIMEOUT
- CIO_HARD_FAIL.

EACCES Indicates that the calling application does not have NET_CONFIG authority.

ENOBUFS Indicates that no buffers are available.

Implementation Specifics

The X25_LINK_DISCONNECT operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The X25_LINK_STATUS operation, X25_LINK_CONNECT operation.

X.25 **ioctl** Operations.

X25_LINK_STATUS x25sioctl Operation (Link Status)

Purpose

Returns the status of the link.

Description

The X25_LINK_STATUS operation returns the status of a link. This operation returns the last known status of the link to the calling program.

For the X25_LINK_STATUS operation, the *arg* parameter points to a buffer. The buffer is filled on return of this operation with a **x25_link_status** structure. This structure contains the following five fields:

status Filled in by an X.25 device handler with a return value when the return code is EIO.

packet Identifies the status of the packet layer. This field has the following possible values:

0	Link disconnected
1	Connecting link
2	Link connected.

frame Specifies the status of the frame layer. This field has the same values as the **packet** field.

physical Specifies the status of the physical layer. This field has the same values as the **packet** field.

no_of_vcs_in_use Identifies the number of virtual circuits currently in use on the link.

Execution Environment

The X25_LINK_STATUS operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

ENOBUFS Indicates that no buffers are available.

EFAULT Indicates that an invalid address was specified.

EIO Indicates that an error occurred. The **status** field of the **x25_link_status** structure contains one of the following common exception codes:

- CIO_HARD_FAIL
- CIO_BAD_MICROCODE.

Implementation Specifics

The X25_LINK_STATUS operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The X25_LINK_DISCONNECT operation, X25_LINK_CONNECT operation.

X.25 **ioctl** Operations.

X25_LOCAL_BUSY x25sioctl Operation (Local Busy)

Purpose

Enables or disables receiving of data packets on a port.

Description

Note: Only the user who called the CIO_START operation can call the X25_LOCAL_BUSY operation.

The X25_LOCAL_BUSY operation enables or disables the receiving of data and interrupt packets on a given session. This operation can be used to slow down large blocks of received data or reduce the number of buffers required. However, clear-and-reset packets are still passed on.

The effects of disabling received packets do not take place immediately after calling the Local Busy operation. Data packets that have arrived before the call, or packets currently being read off the card, are passed on.

The X25_LOCAL_BUSY operation does not affect the outcome of the **x25read** or **x25select** entry points. These operations continue to wait for received packets. To query the status of local busy on a session, use the X25_QUERY_SESSION operation.

Parameter Block

For the X25_LOCAL_BUSY operation, the *arg* parameter points to a buffer that contains the **x25_local_busy** structure. This structure contains the following fields:

busy_mode Specifies the handling of data packets on the session. This field accepts one of the following values:

0	Enables the receiving of data on this session.
1	Disables the receiving of data on this session.

session_id Identifies the session this operation applies to.

Execution Environment

The X25_LOCAL_BUSY operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EINVAL	Indicates that the session ID specified was not valid or the **busy_mode** field was illegal.
EFAULT	Indicates that an invalid address was specified.
EACCES	Indicates that the call must be made by the user who issued the CIO_START operation.

Implementation Specifics

The X25_LOCAL_BUSY operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The X25_QUERY_SESSION operation, CIO_START operation.

X.25 **ioctl** Operations.

X25_QUERY_ROUTER_ID x25sioctl Operation (Query Router ID)

Purpose

Queries an entry in the routing table.

Description

Note: This operation is restricted to user programs that have NET_CONFIG permission.

The X25_QUERY_ROUTER_ID operation queries an entry in the routing table. For this operation, the *arg* parameter points to the **x25_router_query** structure. This structure contains the following fields:

router_id	Specifies what entry to query.
pid	Set on return of the query to the process ID of the listening process. A value of 0 (zero) indicates that no process is listening.

Execution Environment

The X25_QUER_ROUTER_ID operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EFAULT	Indicates that the address specified was not valid.
EINVAL	Indicates that the router ID specified is not in the router table.

Implementation Specifics

The X25_QUER_ROUTER_ID operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

X.25 **ioctl** Operations.

X25_QUERY_SESSION x25sioctl Operation (Query Session)

Purpose

Queries the status of an open X.25 session.

Description

Note: This call only succeeds on switched virtual circuits (SVCs) and permanent virtual circuits (PVCs).

The X25_QUERY_SESSION supplies the information for a session in the user's data area. The packet size, window size, and throughput class values are not available until the session is completely established. To query the static configuration, use the CIO_QUERY operation.

The x25_query_session_data Parameter Block

For the X25_QUERY_SESSION operation the *arg* parameter points to the **x25_query_session_data** structure. Within this structure, the session to be queried is identified either by a nonzero session ID or a nonzero logical channel number. If both the **session_id** and **logical_channel** fields are nonzero, the **session_id** field is used.

The fields in the **x25_query_session_data** structure are set on return. All the X.25 facilities specified by the structure's field are encoded as in the ISO 8208 definition. The **x25_query_session_data** structure contains the following fields:

netid	Identifies the user-defined correlator set by the CIO_START operation.
session_name	Identifies the user-defined name set by the CIO_START operation.
session_id	Identifies the device handler's correlator returned from the CIO_START operation.
local_busy	Contains a value of 1 (one) if the session is in Local Busy mode, or a 0 (zero), if not.
session_protocol	Specifies the higher level protocol specified by the user for this session in the CIO_START operation.
logical_channel	Identifies the X.25 logical channel number used by the session.
tx_tclass	Specifies the transmit throughput-class facility in use on the session. If the call has not been established, this is 0.
rx_tclass	Specifies the receive throughput-class facility in use on the session. If the call has not been established, this is 0.
tx_packet_size	Identifies the outbound packet size in use on the session. If the call has not been established, this field is set to 0.
rx_packet_size	Identifies the inbound packet size in use on the session. If the call has not been established, this field is set to 0.
tx_window_size	Identifies the outbound window size in use on the session. If the call has not been established, this is set to 0.
rx_window_size	Identifies the inbound window size in use on the session. If the call has not been established, this is set to 0.

Execution Environment

The X25_QUERY_SESSION operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EFAULT	Indicates that an invalid address was specified.
EINVAL	Indicates that the session ID was not valid or the **logical_channel** field was not valid.

Implementation Specifics

The X25_QUERY_SESSION operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The CIO_START operation.

X25_REJECT x25sioctl Operation (Reject Call)

Purpose

Provides the means to reject an incoming X.25 call.

Description

Note: A call can be rejected only by the process that called the CIO_START operation.

The X25_REJECT operation is used to reject an X.25 incoming call that was forwarded to a session of type SESSION_SVC_LISTEN. This operation causes a Clear Request to be issued in response to the incoming call.

The X25_REJECT operation returns immediately to the caller, before the command completes. If the immediate return indicates no error, the X.25 device handler builds a status block of type X25_REJECT_DONE on receipt of a Clear Confirm or Clear Indication. For kernel mode processes, the status block is passed to the associated status function. The status function is specified when the X.25 channel is opened. For user-mode processes, the block is placed in the associated status and exception queue.

The x25_reject_data Parameter Block

For the Reject Call operation, the *arg* parameter points to a **x25_reject_data** structure. The **sb.status** field of this structure is meaningful on return only if the return code is EIO.

For the X25_REJECT operation, the *ext* parameter optionally points to a buffer containing the data required for a Clear Request packet. This data is in the form described in the **mbuf** structure. For a kernel-mode process, this parameter points to the **mbuf** structure. For a user-mode process, it points to a buffer of the same format in user space. If the pointer is NULL, the Clear Request is sent with default cause-and-diagnostic codes, and no facilities or user data.

Execution Environment

The X25_REJECT operation can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. If –1 is returned, the kernel sets the **errno** global variable to one of the following values:

EFAULT Indicates that an address was specified that is not valid.

EIO Indicates that an error has occurred. The **arg–>status** field in the **x25_reject_data** structure contains one of four common exception codes:

- CIO_HARD_FAIL
- CIO_NOMBUF
- CIO_TIMEOUT
- CIO_NOT_STARTED.

In addition, the **arg–>status** field may return one of three X.25-specific codes:

X25_BAD_CALL_ID	The **call_id** field specified is not valid.
X25_PROTOCOL	Indicates that a protocol error occurred.
X25_CLEAR	Indicates that the session has been cleared.

EINVAL	Indicates one of the following occurred: • A reject was issued on a session that was not started in SESSION_SVC_LISTEN mode. • The *ext* parameter points to a buffer that does not have a packet type of PKT_CLEAR_REQ.
EACCES	Indicates that the reject must be performed by the same process that called the X.25 CIO_START operation.

Implementation Specifics

The X25_REJECT operation functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sioctl** entry point.

The CIO_START operation.

The **mbuf** structure, X.25 ioctl Operations, Common X.25 Device Handler Structures in *Kernel Extensions and Device Support Programming Concepts*

x25smpx X.25 Device Handler Entry Point

Purpose

Provides the means to allocate and deallocate a channel into X.25 device handler.

Syntax

int x25xmpx (*devno*, *chan*, *channame*)
 dev_t *devno*;
 int **chan*;
 char **channame*;

Parameters

devno	Specifies the major and minor device numbers.
chan	Specifies the channel ID. If the *channame* parameter is NULL, the *chan* parameter identifies the channel to be deallocated. Otherwise, the **x25smpx** entry point returns the ID of the allocated channel to the *chan* parameter.
channame	Points to the remaining path name that describes the channel. This parameter accepts the following five values:

- The parameter is equal to NULL. This value indicates that the channel is to be deallocated.
- The parameter points to a NULL string. This value causes the **x25smpx** entry point to return an ID that allows a normal open sequence.
- The parameter points to D (Diagnostic mode). This value causes the **x25smpx** entry point to return an ID that allows the device handler to be opened in Diagnostic mode.
- The parameter points to M (Monitor mode). This value causes the **x25smpx** entry point to return an ID that allows the device handler to be opened in Monitor mode.
- The parameter points to R (Router mode). This value causes the **x25smpx** entry point to return an ID that allows the device handler to be opened in Router mode.

Description

Note: This entry point is called by the kernel. It cannot be called directly by a user- or kernel-mode process.

The **x25smpx** entry point provides the means for allocating and deallocating a channel into the X.25 device handler. This entry point is called by the kernel in response to an **open** subroutine (before calling the **x25sopen** entry point) or in response to a **close** subroutine. (after calling the **x25sclose** entry point).

Execution Environment

An **x25smpx** entry point can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. The kernel sets the **errno** global variable to one of the following values:

EINVAL	Indicates that a parameter was specified that was not valid.
EPERM	Indicates that an open in the specified mode is denied.
EBUSY	Indicates that the device is already open in Diagnostic, Monitor, or Router mode.

Implementation Specifics

The **x25smpx** entry point functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sclose** entry point, **x25sopen** entry point.

The **open** subroutine, **close** subroutine.

X.25 Device Handler Modes in *Kernel Extensions and Device Support Programming Concepts*.

x25sopen X.25 Device Handler Entry Point

Purpose

Initializes a channel into the X.25 device handler.

Syntax

int x25sopen (*devno*, *devflag*, *chan*, *ext*)
dev_t *devno*;
ulong *devflag*;
int *chan*;
struct kopen_ext **ext*;

Parameters

devno	Specifies the major and minor device numbers.
devflag	Indicates how the device was opened and whether the caller is a user or kernel-mode process. This parameter accepts the following flags:

DKERNEL	A kernel-mode process called the entry point. This flag is clear if a user-mode process called the entry point.
DREAD	Open for reading. This is the default for the X.25 handler regardless of whether this flag is set.
DWRITE	Open for writing. This is the default for the X.25 handler regardless of whether this flag is set.
DAPPEND	Open for appending. The X.25 handler ignores this flag.
DNDELAY	If this flag is set, the X.25 device handler performs nonblocking reads and writes. Otherwise, blocking reads and writes are performed.

chan	Identifies the channel number assigned by the **x25smpx** routine.
ext	Specifies the extended system call parameter. This parameter is required for kernel-mode processes and ignored for user mode processes.

Description

The **x25sopen** entry point performs data-structure allocation and initialization. Time-consuming tasks, such as port initialization and connection establishment, are deferred until the first CIO_START operation is issued. This call is synchronous and does not return until the **x25open** entry point is complete.

Note: If this is the first open request to the X.25 device handler, the interrupt level and interrupt handler entry point are registered.

Parameter Block

For the **x25sopen** entry point, the *ext* parameter can be a pointer to the **kopen_ext** structure defined in the **<sys/comio.h>** header file. This structure contains the following five fields:

status	Identifies the status of the open process. This value is meaningful only if the code EIO is returned.
open_id	Specifies the channel correlator for kernel mode processes. This value is passed to kernel functions to identify on which channel an event occurred.
rx_fn	Specifies the address of a kernel procedure. This procedure is called by the X.25 device handler whenever received data is to be processed. This kernel procedure must be defined as follows:

void *rx_fn* **(***open_id*, *read_ext*, *mbufptr***)**
ulong *open_id*;
struct x25_read_ext **read_ext*;
struct mbuf **mbufptr*;

The parameters in this kernel procedure are defined as follows:

open_id	Specifies the ID of this instance of the **x25sopen** entry point The device handler sets this parameter to the ID originally passed to the X.25 device handler with the **x25sopen** entry point.
read_ext	Contains the status of the **x25sopen** entry point. Currently, this parameter accepts a value of CIO_OK or CIO_BUF_OVFLW.
mbufptr	Points to received data. This data is in the form described by the **mbuf** structure.

The kernel-mode process making the call to the **x25sopen** entry point is responsible for pinning the **rx_fn** kernel procedure before making the call. When the X.25 device handler calls the kernel procedure, the X.25 device handler pins the **mbuf** structure. It is the responsibility of the **rx_fn** kernel procedure to free the pinned **mbuf** structure.

tx_fn Identifies the address of a kernel procedure. The X.25 device handler calls this procedure when both the following conditions are true:

- The most recent **x25swrite** entry point for this channel failed with a return code of EAGAIN, indicating the write request was not performed.
- The **x25sopen** entry point, or the most recent **x25sioctl** operation for this channel, indicates Nonblocking mode (DNDELAY) is set.

The **tx_fn** kernel process should be defined as follows:

void *tx_fn* **(***open_id***)**
ulong *open_id*;

The parameter in this kernel process is defined as follows:

open_id Identifies the ID of the **x25sopen** entry point. The device handler sets this value to the ID passed with the **x25sopen** entry point.

The kernel-mode process making the call to the **x25sopen** entry point is responsible for pinning the **tx_fn** kernel procedure before making the call.

stat_fn The address of a kernel procedure to be called by the X.25 device handler whenever a status block becomes available. The kernel procedure should have the following structure:

void *stat_fn* (*open_id, sblk_ptr*)
ulong *open_id*;
struct status_block **sblk_ptr*;

The kernel procedure parameters have the following values:

open_id	Identifies the ID of the open entry point. The device handler sets this value to the ID passed with the **x25sopen** entry point.
sblk_ptr	Points to a status block.

The kernel-mode process that calls the **x25sopen** entry point is responsible for pinning the **stat_fn** kernel procedure before making the open call.

The **rx_fn** procedure, **tx_fn** procedure, and **stat_fn** kernel procedure are all made synchronously at high priority. It is therefore imperative that the called kernel procedure return quickly. Until the return, the kernel procedure cannot call any other device entry point.

Note: Entry points are associated with a channel initialized by the **x25sopen** entry point. Sessions are initialized by the CIO_START operation. A single channel supports numerous sessions.

Execution Environment

An **x25sopen** entry point can be called from the process environment only.

Return Values

A return code of –1 indicates an unsuccessful operation. The kernel sets the **errno** global variable to one of the following values:

EINVAL Indicates that a kernel user passed a function that is not valid.

EIO Indicates that an error has occurred. The **sb.status** field contains the CIO_HARD_FAIL return value indicating a hardware failure was detected.

EINTR Indicates that the **open** subroutine was interrupted.

ENODEV Indicates that the device requested does not exist.

EBUSY Indicates that the maximum number of opens was exceeded. This error is received if an attempt is made to open the channel in Diagnostic mode while there are other open channels on the minor device number. In addition, this error is received if an attempt is made to open a channel where one is already open and running in Monitor or Router mode.

ENOMEM Indicates that the X.25 device handler was unable to allocate space required for the open.

ENXIO Indicates that one of the following occurred:

- An attempt was made to open the X.25 device handler before it was configured.
- The interrupt could not be registered.

Implementation Specifics

The **x25sopen** entry point functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The CIO_START operation.

The **open** subroutine.

The **x25swrite** entry point, **x25smpx** entry point, **x25sioctl** entry point.

List of Common Status/Exception Codes, X.25 Device Handler Modes, The X.25 mbufs Structure in *Kernel Extensions and Device Support Programming Concepts*.

x25sread X.25 Device Handler Entry Point

Purpose

Provides the means to receive data from the X.25 adapter.

Syntax

int x25sread (*devno*, *uiop*, *chan*, *ext*)
dev_t *devno*;
struct uio **uiop*;
int *chan*;
struct x25_read_ext **ext*;

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure.
chan	Identifies the channel number assigned by the **x25smpx** routine.
ext	Points to the **x25_read_ext** structure. This structure is found in the **<sys/x25user.h>** header file and it contains a **call_id** field and a **re.status** field. The **call_id** field is only valid on sessions of type SESSION_SVC_LISTEN. The **re.status** field is meaningful only if the return value is EIO.

Description

Note: This entry point can only be called by user mode processes. Data received for a kernel-mode process is passed to the **rx_fn** kernel procedure specified by the **x25sopen** entry point.

The **x25sread** entry point provides the means to receive incoming data on the session specified by **session_id** field. If the **session_id** field is 0 (zero) and the device was opened in normal mode, data for any session started by this channel is returned, and the **session_id** field is filled in accordingly. The X.25 device handler copies the data to the user buffer and decrements the **uiop–>resid** field by the number of bytes moved.

X.25 data is made up of an m-bit sequence. This sequence is consolidated before it is made available for read operations. The exception is sessions of type X25_SESSION_YBTS. For these sessions, each packet is available as a separate data block.

Notes:

1. The order of incoming data is preserved for each session, but is not guaranteed across sessions.
2. The **x25_packet_data** common data structure describes the buffering of incoming X.25 packet sequences. This structure is found in the **<sys/x25user.h>** header file.

The **x25sread** entry point can be a blocking or nonblocking read. The type of read is determined by flags specified by the **x25sopen** entry point when the channel is opened. If the read is blocking, and no data is available, the **x25sread** entry point blocks until data is received. If the read is nonblocking and no data is available, the entry point returns an error code.

If the current session was initialized for listening, the only data that can be read on the session is an incoming call. The user process should respond by issuing a X25_REJECT operation on the current session or by starting a new session with a Start Session CIO_START operation, to accept the call.

When a PKT_CLEAR_IND packet is received, the user must respond with a CIO_HALT operation. As a result, no further **x25swrite** entry points are accepted. If the session is a SESSION_MONITOR type, then the data buffer contains monitor control sequences.

Parameter Block

For the **x25sread** entry point, the *arg* parameter returns a pointer to the **uio** structure. This structure specifies the location and length of the caller's data area to transfer information to. The **uio** structure is defined in the **<sys/uio.h>** header file.

The data is in the form described in the **mbuf** structure. The value for the **packet_type** field for SESSION_SVC_LISTEN sessions is PKT_INCOMING_CALL. For other sessions, the possible packet types are the following:

- PKT_DATA
- PKT_INT
- PKT_INT_CONFIRM
- PKT_RESET_IND
- PKT_RESET_CONFIRM
- PKT_D_BIT_ACK
- PKT_CLEAR_IND (except for sessions of type SESSION_PVC).

Execution Environment

The **x25sread** entry point can be called from the process environment only.

Return Values

EFAULT	Indicates a buffer area that is not valid.
EINVAL	Indicates a parameter that is not valid.
EIO	Indicates that an error has occurred. The **ext->status** field in the **x25_read_ext** structure contains one of the following values: • CIO_NOT_STARTED • CIO_HARD_FAIL • CIO_LOST_DATA.
EMSGSIZE	Indicates that the buffer was not large enough to receive the packet data. The receiver data is preserved within the device driver until a read is issued with a large enough buffer.
EAGAIN	Indicates that there were no packets to be read and the device was opened with the **DNDELAY** flag set.

Implementation Specifics

The **x25sread** entry point functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25swrite** entry point, **x25sopen** entry point, **x25smpx** entry point.

The CIO_START operation, CIO_HALT operation, X25_REJECT operation.

The **mbuf** structure.

Sessions with the X.25 Device Handler, Data Transmission for the X.25 Device Handler, Data Reception for the X.25 Device Handler in *Kernel Extensions and Device Support Programming Concepts.*

x25sselect X.25 Device Handler Entry Point

Purpose

Determines if a specified event occurred on a device.

Syntax

int x25sselect (*devno*, *events*, *reventp*, *chan*)
dev_t *devno*;
ushort *events*;
ushort **reventp*;
int *chan*;

Parameters

devno	Specifies the major and minor device numbers.
events	Identifies what events to check. The *events* parameter is indicated by a bitwise OR using the following flags:

DPOLLIN	Check if the receive data is available.
DPOLLOUT	Check if transmission is possible. For the X.25 device handler, this event is always true.
DPOLLPRI	Check if status is available.
DPOLLSYNC	The request is synchronous. The **x25sselect** entry point should *not* perform a **selnotify** kernel service if the events occur later.

reventp	Returns the events pointer. The **x25sselect** entry point uses this parameter to indicate which of the selected events are true at the time of the call. The *reventp* parameter is indicated by a bitwise OR of the **DPOLLIN**, **DPOLLOUT**, or **DPOLLPRI** flag, as appropriate.
chan	Identifies the channel number assigned by the **x25smpx** entry point.

Description

Note: This entry point should only be called by user mode processes using the **select** or **poll** subroutine.

The **x25sselect** entry point determines if a specified event occurred on a device. If one or more events specified by the *events* parameter are true, this entry point updates the *reventp* parameter by setting the corresponding bits.

If none of the events are true, the *reventp* parameter is set to 0 (zero) and the entry point checks the **DPOLLSYNC** flag. If this flag is set, the request is synchronous and the entry point simply returns. If this flag is false, the **x25sselect** entry point records which events were requested. When one or more of the events subsequently becomes true, the **x25sselect** entry point calls the **selnotify** kernel subroutine to notify the user process.

When the X.25 device handler is in a state that prevents any of the events from being satisfied (such as an adapter failure), then the **x25sselect** entry point sets the *reventp* parameter to 1 (one) for the appropriate event. This prevents the **select** or **poll** subroutine from waiting indefinitely.

Note: Unless the session protocol is **PROTOCOL_YBTS**, an X.25 packet sequence cannot satisfy a **x25sselect** entry point until the final packet of the sequence is received or the sequence is otherwise terminated (for example, by the arrival of a clear indication).

Execution Environment

An **x25sselect** entry point can be called from the process environment only.

Return Value

A return code of –1 indicates an unsuccessful operation. The kernel sets the **errno** global variable to the following value:

EINVAL	Indicates that an invalid argument was specified or that the **x25sselect** entry point was called by a kernel mode user.

Implementation Specifics

The **x25sselect** entry point functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **select** subroutine, **poll** subroutine.

The **selnotify** kernel service.

Data Transmission for the X.25 Device Handler, Data Reception for the X.25 Device Handler in *Kernel Extensions and Device Support Programming Concepts.*

x25swrite X.25 Device Handler Entry Point

Purpose

Provides the means to send data to the X.25 adapter.

Syntax

int x25swrite (*devno*, *uiop*, *chan*, *ext***)**
dev_t *devno*;
struct uio **uiop*;
int *chan*;
struct x25_write_ext **ext*;

Parameters

devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure.
chan	Identifies the channel number assigned by the **x25smpx** routine.
ext	Points to the **struct x25_write_ext** structure. This structure is found in the **<sys/x25user.h>** header file.

Description

Note: Call-establishment or termination packets cannot be sent using this entry point.

The **x25swrite** entry point provides the means to send a X.25 data packet to the adapter.

uio Structure

For the **x25swrite** entry point, the *uiop* parameter is a pointer to a **uio** structure. This structure is described in the **<sys/uio.h>** file. The **uio** structure specifies the location and length of the caller's data.

This routine checks the **uiop–>segflag** field to determine whether the data is in user space or kernel space. If the data is in kernel space, then the **uiop–>uio_ iov–>uio_base** field points to an **mbuf** structure chain containing the data for transmission. If the data is in user space, then the **uiop–>uio_iov** field points to an array of **iovec** structures.

The data is in the form described by the **mbuf** structure. For a kernel-mode process, the **mbufs** structure containing the data should be pinned before making this call.

For session types of SESSION_SVC_OUT, SESSION_SVC_IN, or SESSION_PVC, the possible values for the **packet_type** field are the following:

- PKT_DATA
- PKT_INT
- PKT_INT_CONFIRM
- PKT_RESET_REQ
- PKT_RESET_CONFIRM
- PKT_D_BIT_ACK.

Note: For a SESSION_MONITOR session, the **packet_type** field must have a value of PKT_MONITOR.

If the value of the **packet_type** field is PKT_DATA and the data buffer is larger than the packet size, the data is transmitted as an X.25 packet sequence.

If a previous incoming data packet has been received with the D bit set, the incomming packet must be acknowledged with a PKT_D_BIT_ACK packet type before any further packets can be accepted by the device handler for this session.

The x25_write_ext Parameter Block

For the **x25swrite** entry point, the *ext* parameter points to the **x25_write_ext** parameter block. The **x25_write_ext** structure contains a **write_extension** (**we**) structure and a **session_id** field.

The **we.flag** field consists of the bitwise OR of one or more of the following values:

CIO_NOFREE_MBUF
Setting this bit causes the X.25 device handler to retain the **mbuf** structure after transmission is complete. If a kernel-mode process sets this bit, it must also do the following:

1. Determine when the X.25 device handler is finished with the **mbuf** structure.
2. Free the **mbuf** structure.

For a user-mode process, the device handler always frees the **mbuf** structure.

CIO_ACK_TX_DONE
Setting this bit causes the X.25 device handler to acknowledge completion by building a CIO_TX_DONE status block for the caller when the write is complete.

The **we.status** field is meaningful only if the return value is EIO.

Execution Environment

The **x25swrite** entry point can be called from the process environment only.

Return Values

EINVAL Indicates one of the following:

- A parameter was used that is not valid.
- A write was made on a listen session.

EIO Indicates that an error has occurred. The **ext->status** field contains one of the following common exception codes:

- CIO_NOT_STARTED
- CIO_HARD_FAIL
- CIO_NOMBUF
- CIO_TIMEOUT.

Otherwise, the field contains one of the following X.25 specific codes:

X25_NO_ACK	A data packet with the D bit set requires acknowledgment. Data packets cannot be sent until the acknowledgment is completed.
X25_NO_ACK_REQ	A PKT_D_BIT_ACK was sent and no packets required a D bit acknowledgment.
X25_PROTOCOL	A protocol error occurred.

X25_RESET	The session is in reset state; the data packet could not be sent.
X25_CLEAR	The session has been cleared.
X25_NO_LINK	The X.25 link is not established.
X25_BAD_PKT_TYPE	The **packet_type** field passed in the *uiop* parameter block is invalid for the session type.

EAGAIN Indicates that the transmit queue is full and the **DNDELAY** flag is set. The command was not accepted.

Implementation Specifics

The **x25swrite** entry point functions with an IBM X.25 Interface Co-Processor/2 that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Related Information

The **x25sread** entry point.

Common X.25 Device Handler Structures, Sessions with the X.25 Device Handler in *Kernel Extensions and Device Support Programming Concepts.*

Chapter 6. High Function Terminal (HFT) Subsystem

HFCHGLOC ioctl Operation (Change Locator)

Purpose

Changes the attributes of the mouse or tablet locator.

Description

The Change Locator (HFCHGLOC) **ioctl** operation allows you to change the following aspects of the mouse or tablet:

- Mouse sample rate
- Mouse resolution
- Mouse thresholds
- Mouse scaling factor
- Tablet sample rate
- Tablet resolution
- Tablet dead zones
- Tablet conversion
- Tablet origin.

The HFCHGLOC **ioctl** operation is invoked by:

```
int ioctl(fildes, HFCHGLOC, &arg)
int fildes;
struct hfchgloc arg;

struct hfchgloc {
        ulong hf_cmd;
        ushort loc_value1;
        ushort loc_value2;
        };
```

The **hf_cmd** field tells which attribute to set, and the **hf_value1** and **hf_value2** fields give the value of this attribute. The **hfchgloc** structure is defined in the **sys/hft.h** header file.

Use of the fields in the **hfchgloc** structure is described in Change Locator (HFCHGLOC) in **hft.h** File Structures for Special **ioctl** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Change Locator (HFCHGLOC) in **hft.h** File Structures for Special **ioctl** Operations in *Files Reference.*

Understanding Keyboard, Tablet, Sound, and Mouse Devices, HFT ioctl Operations, HFT Device Driver (HFTDD) User Interface Overview in *Kernel Extensions and Device Support Programming Concepts.*

HFCMON ioctl Operation (Exit Monitor Mode)

Purpose

Causes the display system to exit Monitor mode (MOM).

Description

The Exit Monitor Mode (HFCMON) **ioctl** operation releases Monitor mode. There are no structures required to exit Monitor mode.

The Exit Monitor Mode (HFCMON) **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFCMON, 0)
int fildes;
```

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Enter and Exit Monitor Mode (HFSMON) and (HFCMON) in **hft.h** File Structures for Special **ioctl** Operations in *Files Reference*.

Enter Monitor Mode (HFSMON) **ioctl** Operation.

HFT Device Driver (HFTDD) User Interface Overview, Understanding Monitor Mode, How to Exit Monitor Mode, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts*.

HFDSOUND ioctl Operation (Disable Sound Signal)

Purpose

Informs a virtual terminal of the intent to discontinue the use of sound signals.

Description

The Disable Sound Signal (HFDSOUND) **ioctl** operation informs the virtual terminal of the intent to discontinue the use of sound signals. Sound-response signals are not sent.

This operation merely disables the routing of sound-response signals. It does not affect the ability to use the sound speaker itself.

The Disable Sound Signal HFDSOUND **ioctl** operation is invoked as follows:

```
int ioctl (fildes, HFDSOUND, 0)
int fildes;
```

No structures are required to disable sound signals.

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Enable and Disable Sound Signals (HFESOUND and HFDSOUND) in **hft.h** File Structures for Special **ioctl** Operations in *Files Reference*.

Enable Sound Signal **ioctl** Operation.

HFT Device Driver (HFTDD) User Interface Overview, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts*.

HFESOUND ioctl Operation (Enable Sound Signal)

Purpose

Enables the routing of the sound-response signal.

Description

The Enable Sound Signal (HFESOUND) **ioctl** operation informs the virtual terminal of the intent to use sound signals.

The application is notified that the sound is complete if:

- A sound request later occurs (because the user sends a Send Sound **write** operation)
- The **hfsound** structure has the **HFSIGSOUND** flag set in the **hf_mode** field.

This action enables the routing of the sound response signals. It does not affect the ability to use the sound speaker itself, but merely determines whether the user is notified by signal that the sound has occurred.

The Enable Sound Signal **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFESOUND, &arg)
int fildes;
struct hfsmon arg;

struct hfsmon
{
  int hf_momflags;
};
```

The caller can use the **hfsmon** structure to indicate whether one or all members of the current process group are to receive a sound response signal. The **hfsmon** structure is defined in the **sys/hft.h** header file. For more information on using this structure, see Enable and Disable Sound Signal (HFESOUND and HFDSOUND) in **hft.h** File Structures for Special **ioctl** Operations.

The HFESOUND operation is valid in both KSR and MOM modes.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Enable and Disable Sound Signals (HFESOUND and HFDSOUND) in **hft.h** File Structures for Special **ioctl** Operations in *Files Reference.*

Disable Sound Signal **ioctl** Operation.

HFT Device Driver (HFTDD) User Interface Overview, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFESWKBD ioctl Operation (Enable Software Keyboard)

Purpose

Enables the software keyboard map to be used by a virtual terminal.

Description

The Enable Software Keyboard (HFESWKBD) **ioctl** operation allows you to change the software keyboard map currently used by a virtual terminal to another keyboard map. This keyboard map must currently be in the system (after being added by the HFADDSWKBD option of the HRFCONF **ioctl** operation). Use the HFQUERY **ioctl** operation to determine the available keyboard maps.

The HFESWKBD **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFESWKBD, &kbdid)
int fildes;
long *kbdid;
```

The **kbdid** field indicates the software keyboard to be enabled and can be determined from the **hf_kbdid** field returned in the HFQUERY **ioctl** operation.

The software keyboard is represented by the **vtmdkey** structure as defined in the **hft.h** header file. This structure is described in

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Enable Software Keyboard (HFESWKBD) in **hft.h** File Structures for Special **ioctl** Operations in *Files Reference.*

Query Software Keyboards Query **ioctl** Operation.

The Reconfigure (HRFCONF) **ioctl** Operation.

HFT Device Driver (HFTDD) User Interface Overview, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFQERROR ioctl Operation (Query I/O Error)

Purpose

Returns a detailed device error code for operations that have failed due to a High Function Terminal (HFT) subsystem failure.

Description

The Query I/O Error (HFQERROR) **ioctl** operation returns a detailed device error code. If an I/O operation or other operation to the HFT fails because of an HFT subsystem error, the system call returns a nonzero value and sets the **errno** global variable to EIO.

The calling program can get a more detailed device error code by using the **ioctl** operation to issue an HFQERROR operation.

The HFQERROR **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFQERROR, &arg)
int fildes;
long arg;
```

The *arg* value from the HFQERROR **ioctl** operation is either 0 (indicating that the last I/O operation was successful) or the detailed error code.

Error Codes

The possible terminal error codes are:

500 =	LPFK/dial device driver failed.
501 =	HFT device driver failed.
502 =	Keyboard device driver failed.
503 =	Mouse device driver failed.
504 =	Sound device driver failed.
505 =	Screen manager failed.
506 =	Tablet device driver failed.
507 –	Display driver initialization failed.
508 =	Display driver activate failed.
509 =	Display driver define-cursor failed.
510 =	Display driver terminate failed.
512 =	Virtual device driver failed.
520 =	Display driver deactivate failed.
521 =	Display driver failed while moving lines.
522 =	Display driver failed while clearing a box on screen.

523 =	Display driver failed while copying part of a line.
525 =	Display driver failed while moving cursor position.
526 =	Display driver failed while reading a line segment.
528 =	Display driver failed while writing a string of characters.
529 =	Display driver failed while scrolling text on the display.
530 =	Display driver failed while setting mode.
531 =	Display driver failed while changing color mappings.
997 =	Utility defined in the HFT subsystem failed.
998 =	Kernel function used by the HFT failed.
1000 =	Invalid virtual terminal ID was given.
1001 =	Invalid HFT was given.
1002 =	Maximum number of virtual terminals are already open.
1003 =	Invalid address was given.
1004 =	Failure in sending MOM signal.
1005 =	Failure in sending SAK sequence.
1006 =	Failure in sending **SIGKILL** signal.
1402 =	Failure of **malloc** operation for the required function.
1403 =	Device rejected request.
1404 =	Virtual terminal cannot be deactivated.
1405 =	Invalid request made for this virtual terminal.
1406 =	Invalid device ID was given.
1407 =	Invalid object was given.
1408 =	Device not configured.
1409 =	No virtual terminals open.
1413 =	Bad screen manager process creation.
1477 =	Screen manager command virtual terminal already exists.
1483 =	One or more of the involved virtual terminals are trusted.
1485 =	Screen manager request has invalid virtual terminal ID.
1489 =	Request for an invalid screen manager operation.
1521 =	Invalid length given in VTD.

1522 =	Invalid major type given in VTD.
1523 =	Invalid minor type given in VTD.
1529 =	Device not available.
1530 =	Presentation space size zero.
1531 =	Invalid final character on SGR.
1533 =	Bad font IDs.
1534 =	Bad font size.
1535 =	No font available.
1539 =	Device not enabled.
1541 =	Data not aligned.
1542 =	No cursor exists.
1544 =	Invalid mode.
1545 =	Terminal not active.
1551 =	Invalid parameter.
1552 =	Invalid control sequence.
1700 =	Input to the virtual terminal has been destroyed.
1701 =	Invalid function encountered.
1703 =	Cross memory move failure.
1704 =	MOM buffer overflow.
1705 =	Virtual terminal received a bad queue element.
1800 =	Query to the virtual terminal was incomplete.
1801 =	Response buffer overflow.
1802 =	Invalid parameter given in the query.
1961 =	Invalid selector given in the change-structure request.
1962 =	Bad echo length.
1963 =	Undefined function ID.
1964 =	Predefined key position that cannot be remapped.
1965 =	Invalid key flags given in change structure request.
1966 =	Invalid key position given in change structure request.
1967 =	Bad KSR buffer length.

1968 =	Bad keyboard length.
1970 =	Invalid keyboard state given in change structure request.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Query I/O Error (HFQERROR) in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

HFT Device Driver (HFTDD) User Interface Overview, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts*.

HFQUERY ioctl Operation (Query)

Purpose

Returns information about a virtual terminal or the overall state of the HFT subsystem.

Description

The Query (HFQUERY) **ioctl** operation supports several types of queries about the current virtual terminal.

The HFQUERY **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFQUERY, &arg)
int fildes;
struct arg;
```

Structures for each query command are defined in the **hft.h** header file. Their use is described in **hft.h** File Structures for Query **ioctl** Operations.

Query Command Options

The following queries can be made using the Query HFQUERY **ioctl** operation:

- Query Physical Display IDs
- Query Physical Device
- Query Mouse
- Query Tablet
- Query LPFKs
- Query Dials
- Query Presentation Space
- Query Software Keyboards
- Query HFT Device
- Query Keyboard Status
- Query Retract Device ID.

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Query I/O Error (HFQERROR) in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

HFT Device Driver (HFTDD) User Interface Overview, HFT ioctl Operations*Kernel Extensions and Device Support Programming Concepts.*

HFQUERY ioctl Option: Query Dials

Description

The Query Dials HFQUERY operation returns device information about the dials. This information consists of the number of dials on the device and the granularity of each dial.

The Query Dials HFQUERY **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFQUERY, &arg)
int fildes;
struct hfqdial_lpfk arg;

struct hfqdial_lpfk  {
    struct hfqgraphdev *hf_cmd;
    long   hf_cmdlen;
    struct hfdial_lpfk *hf_resp;
    long   hf_resplen;
};
```

The fields in the **hfqdial_lpfk**, **hfqgraphdev**, and **hfdial_lpfk** structures are defined in the **sys/hft.h** header file. Their use is described in Query Dials Command in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Related Information

Query Dials Command in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFQUERY ioctl Option: Query HFT Device

Description

The Query HFT Device HFQUERY operation returns information about the physical display attributes and keyboard being used by a specified virtual terminal.

The Query HFT Device HFQUERY **ioctl** operation is invoked by the following:

```
int ioctl(fildes, HFQUERY, &arg)
int fildes;
struct hfqhft arg;

struct hfqhft  {
    struct hfqhftc *hf_cmd;
    long   hf_cmdlen;
    struct hfqhftr *hf_resp;
    long   hf_resplen;
};
```

The fields in the **hfqhft**, **hfqhftc**, and **hfqhftr** structures are defined in the **/sys/hft.h** header file. Their use is described in Query HFT Device Command in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Related Information

Query HFT Device Command in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFQUERY ioctl Option: Query Keyboard Status

Description

The Query Keyboard Status HFQUERY operation returns status information about the state of a Japanese (106-key) keyboard.

The Query Keyboard Status HFQUERY **ioctl** operation is invoked by the following:

```
int ioctl(fildes, HFQUERY, &arg)
int fildes;
struct hfqkbs arg;

struct hfqkbs  {
    struct hfqkbsc *hf_cmd;
    long   hf_cmdlen;
    struct hfqkbsr *hf_resp;
    long   hf_resplen;
};
```

The fields in the **hfqkbs**, **hfqkbsc**, and **hfqkbsr** structures are defined in the **/sys/hft.h** header file. Their use is described in Query Keyboard Status Command in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Related Information

Query Keyboard Status Command in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFQUERY ioctl Option: Query Lighted Programmable Function Keys

Description

The Query Lighted Programmable Function Keys HFQUERY operation returns device information about the LPFKs. This information consists of the number of LPFKs on the device, and whether each LPFK is enabled or not.

The Query LPFK **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFQUERY, &arg)
int fildes;
struct hfqdial_lpfk arg;

struct hfqdial_lpfk  {
    struct hfqgraphdev *hf_cmd;
    long   hf_cmdlen;
    struct hfdial_lpfk *hf_resp;
    long   hf_resplen;
};
```

The fields in the **hfqdial_lpfk**, **hfqgraphdev**, and **hfdial_lpfk** These structures are defined in the **sys/hft.h** header file. Their use is described in Query Lighted Programmable Function Keys (LPFKs) Command in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Related Information

Query Lighted Programmable Function (LPFKs) Key Command in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference*.

HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts*.

HFQUERY ioctl Options: Query Mouse and Query Tablet

Description

The Query Mouse and Query Tablet HFQUERY operations return extensive information about the mouse or tablet device attributes for a virtual terminal. These attributes include resolution and sample rates.

The Query Mouse and Query Tablet HFQUERY **ioctl** operations are invoked as follows:

```
int ioctl(fildes, HFQUERY, &arg)
int fildes;
struct hfqloc arg;

struct hfqloc {
    struct hfqgraphdev *hf_cmd;
    long   hf_cmdlen;
    struct hfqlocr     *hf_resp;
    long   hf_resplen;
};
```

The fields in the **hfqloc**, **hfqgraphdev** and **hfqlocr** structures are defined in the **sys/hft.h** header file. Their use is described in Query Mouse Command in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Related Information

Query Mouse Command in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFQUERY ioctl Option: Query Physical Device

Description

The Query Physical Display HFQUERY operation returns information about displays and locator devices. It fills the response buffer with information about the following:

- Mouse and tablet
- Physical display device
- Physical display font information
- Physical display color information.

The Query Physical Display HFQUERY **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFQUERY, &arg)
int fildes;
struct hfqphdev arg;

struct hfqphdev  {
    struct hfqphdevc *hf_cmd;
    long   hf_cmdlen;
    struct hfqphdevr *hf_resp;
    long   hf_resplen;
};
```

The fields in the **hfqphdev**, **hfqphdevc**, and **hfqphdevr** structures are described in more detail in the **sys/hft.h** header file. Their use is described in Query Physical Device Command in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Related Information

Query Physical Device Command in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFQUERY ioctl Option: Query Physical Display IDs

Description

The Query Physical Display IDs HFQUERY operation responds with information about the display devices: the ID of each device and whether it is present and functional on the system.

The Query Physical Display IDs HFQUERY **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFQUERY, &arg)
int fildes;
struct hfqdevid arg;

struct hfqdevid  {
        struct hfqdevidc *hf_cmd;
        long   hf_cmdlen;
        struct hfqdevidr *hf_resp;
        long   hf_resplen;
};
```

The fields in the **hfqdevid**, **hfqdevidc**, and **hfqdevidr** structures are defined in the **sys/hft.h** header file. Their use is described in Query Physical Display IDs Command in **hft.h** File Structures for Query **ioctl** Operations.

Related Information

Query Physical Display IDs Command in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference*.

HFQUERY ioctl Option: Query Presentation Space

Description

The Query Presentation Space HFQUERY operation returns an ASCII data stream image of the current display screen. All or part of the screen can be queried. Attribute and character set information on the queried block are returned. The HFQUERY operation is valid only in KSR mode and is useful only with HFT-supported fonts.

The Query Presentation Space HFQUERY **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFQUERY, &arg)
int fildes;
struct hfqpres arg;

struct hfqpres  {
    struct hfqpresc *hf_cmd;
    long   hf_cmdlen;
    struct hfqpresr *hf_resp;
    long   hf_resplen;
};
```

The fields in the **hfqpres**, **hfqpresc**, and **hfqpresr** structures are defined in the **sys/hft.h** header file. Their use is described in Query Presentation Space Command in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Related Information

Query Presentation Space Command in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference*.

Understanding Keyboard Send-Receive (KSR) Mode, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts*.

HFQUERY ioctl Option: Query Retract Device ID

Description

The Query Retract Device ID HFQUERY operation returns the virtual terminal ID and the physical display device ID of the virtual terminal being retracted.

Structure

The Query Retract Device ID **ioctl** operation is invoked by:

```
int ioctl (fildes, HFQUERY, &arg)
int fildes;
struct hfqretract  arg;
struct hfqretract {
  struct hfqretractc *hf_cmd;
  long hf_cmdlen;
  struct hfqretractr *hf_resp;
  long hf_resplen;
};
```

The **hfqretract** structure is defined in the **sys/hft.h** header file. Use of the fields in these structures is described in Query Retract Device ID Command in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Related Information

Query Retract Device ID Command in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFQUERY ioctl Option: Query Software Keyboard

Description

The Query Software Keyboard HFQUERY operation returns status information about the software keyboard maps loaded in the High Function Terminal (HFT).

The Query Software Keyboard **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFQUERY, &arg)
  int fildes;
  struct hfqswkb arg;

struct hfqswkb {
  struct hfqswkbc *hf_cmd;
  long hf_cmdlen;
  struct hfqswkbr *hf_resp;
  long hf_resplen;
};
```

The fields in the **hfqswkb**, **hfqswkbc**, and **hfqswkbr** structures are defined in the **sys/hft.h** header file. Their use is described in Query Software Keyboards Command in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Related Information

Query Software Keyboards Command in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFSJKBD ioctl Operation (Set Japanese Keyboard)

Purpose

Updates the software keyboard being used by a virtual terminal, specifically the 106-key keyboard.

Description

The Set Japanese Keyboard (HFSJKBD) **ioctl** operation is designed to work only with the 106-key Japanese keyboard and Japanese licensed program software.

The HFSJKBD **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFSJKBD, &arg)
int fildes;
struct hfbuf arg;
struct hfbuf
{
   struct hfkeymap *hf_bufp;
   int   hf_buflen;
};
```

The structures used in the HFSJKBD **ioctl** operation are the same as those used with the Set Keyboard Map (HFSKBD) **ioctl** operation. The **hf_bufp** field points to an **hfkeymap** structure, and the **hf_buflen** field contains the length of the **hfkeymap** structure. The **hfkeymap** structure is defined in the **sys/hft.h** header file. Use of the fields in this structure is described in Set Keyboard Map (HFSKBD) in **hft.h** File Structures for Special **ioctl** Operations.

Interpretation of Japanese Keyboard Shift States

The HFSJKBD operation differs from the HFSKBD operation in its interpretation of the shift states. The Japanese keyboard has these 6 shift states:

- Roma-ji base
- Roma-ji shift
- Control
- Alternate
- Kana base
- Kana shift.

For base and shift, the distinction between Roma-ji and Kana script systems is determined by the Alt Gr bit in the **hf_kstate** field of the **hfkey** structure. This bit can be combined with the base or shift bit to indicate one of four states. When this bit is 0, Roma-ji is assumed. When this bit is 1, Kana is assumed.

Note: The Japanese keyboard does not have an Alt Gr state.

The **hfkeymap** structure contains an **hf_kstate** field subdivided into groups of bits. One of these groups, the HFSHFMASK bits, specifies shift states that apply to the key being mapped. Six literals have been defined for the HFSHFMASK bits:

HFSHFCTRL	Specifies the Ctrl state.
HFSHFALT	Specifies the Alt state.
HFROMBASE	Specifies the base Roma-ji state.
HFROMSHFT	Specifies the shift Roma-ji state.
HFKANBASE	Specifies the base Kana state.
HFKANSHFT	Specifies the shift Kana state.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Set Keyboard Map (HFSKBD) **ioctl** Operation.

Set Keyboard Map (HFSKBD) in **hft.h** File Structures for Special **ioctl** Operations in *Files Reference.*

HFT Device Driver (HFTDD) User Interface Overview in *Kernel Extensions and Device Support Programming Concepts.*

HFSKBD ioctl Operation (Set Keyboard Map)

Purpose

Updates the software keyboard map being used by a virtual terminal.

Description

The Set Keyboard Map (HFSKBD) **ioctl** operation allows the remapping of most keys on the keyboard. Remapping changes the character or control sequence each key generates when pressed.

The HFSKBD **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFSKBD, &arg)
int fildes;
struct hfbuf arg;
struct hfbuf
{
   struct hfkeymap *hf_bufp;
   int   hf_buflen;
};
```

The **hf_bufp** field points to an **hfkeymap** structure, and the **hf_buflen** field contains the length of the **hfkeymap** structure. The **hfkeymap** structure is defined in the **sys/hft.h** header file. Use of the fields in this structure is described in Set Keyboard Map (HFSKBD) in **hft.h** File Structures for Special **ioctl** Operations.

Mapping Alt Keys

On the U.S. 101-key keyboard, the left Alt key produces the Alt shift state, and the right Alt key produces the Alt Gr shift state. The default keyboard mapping for the Alt and Alt Gr states is identical for all keys.

If the Japanese 106-key keyboard is attached, then the Alt Gr shift state cannot be used.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Set Japanese Keyboard (HFSJKBD) **ioctl** Operation.

HFT Device Driver (HFTDD) User Interface Overview, Understanding Monitor Mode, How to Enter Monitor Mode, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFSMON ioctl Operation (Enter Monitor Mode)

Purpose

Causes the virtual terminal to enter Monitor mode.

Description

The Enter Monitor Mode (HFSMON) **ioctl** operation allows the display system to enter Monitor mode. Monitor mode provides a program with direct control of the physical display screen.

This HFSMON **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFSMON, &arg)
int fildes;
struct hfsmon arg;

struct hfsmon
{
  int hf_momflags;
};
```

The caller of the Enter Monitor Mode HFSMON operation can use the **hfsmon** structure to specify whether one or all members of the current process group are to receive Monitor mode signals. The **hfsmon** structure is defined in the **sys/hft.h** header file. For more information on using this structure, see Enter and Exit Monitor Mode (HFSMON) and (HFCMON) in **hft.h** File Structures for Special **ioctl** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Enter and Exit Monitor Mode (HFSMON) and (HFCMON) in **hft.h** File Structures for Special **ioctl** Operations in *Files Reference.*

Exit Monitor Mode (HFCMON) **ioctl** Operation.

HFT Device Driver (HFTDD) User Interface Overview, Understanding Monitor Mode, How to Enter Monitor Mode, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFTCSMGR ioctl Operation (Control Screen Manager)

Purpose

Requests the screen manager to manipulate the status of virtual terminals.

Description

The Request Screen Manager (HFTCSMGR) **ioctl** operation requests the screen manager to manipulate the status of virtual terminals. Virtual terminals are linked together in a group called the screen manager ring. This ring allows the user to call successive virtual terminals to the display screen.

The terminal at the head of the ring, called the *active* terminal, is the virtual terminal visible on the screen. The last terminal on the ring is called the *tail.* When a new terminal is opened, that terminal becomes the head of the ring. The file descriptor can be associated with any of the virtual terminals in the HFT.

The HFTCSMGR **ioctl** operation is invoked by the following:

```
int ioctl(fildes, HFTCSMGR, &arg)
int fildes;
struct hftsmgrcmd arg;

struct hftsmgrcmd  {
    int hf_cmd;
    int hf_vtid;
    int hf_vsid;
};
```

The **hfsmgrcmd** structure contains an **hf_cmd** field that you must set to the required operation. Screen manager commands are discussed in Screen Manager Operations. The **hfsmgrcmd** structure itself is defined in the **sys/hft.h** header file. Query Screen Manager (HFTQSMGR) in **hft.h** File Structures for Query **ioctl** Operations gives information on using this structure.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Query Screen Manager (HFTQSMGR) in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

Understanding the Screen Manager Ring, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFTGETID ioctl Operation (Get Virtual Terminal ID)

Purpose

Returns the channel number of the current virtual terminal.

Description

The Get Virtual Terminal ID (HFTGETID) **ioctl** operation returns identification information for the HFT virtual terminal specified by the file descriptor to this **ioctl** operation.

The HFTGETID **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFTGETID, &arg)
int fildes;
struct hftgetid arg;

struct hftgetid  {
    dev_t hf_dev;
    ulong hf_pgrp;
    ulong hf_chan;
};
```

The fields in the **hftgetid** structure are defined in the **sys/hft.h** header file. Their use is described in Get Virtual Terminal ID (HFTGETID) in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Get Virtual Terminal ID (HFTGETID) in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference.*

HFT Device Driver (HFTDD) User Interface Overview, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFTQDEV ioctl Operation (Query Device)

Purpose

Obtains information about input and output devices used by the HFT subsystem.

Description

The Query Device (HFTQDEV) **ioctl** operation obtains detailed device information about the device types associated with the HFT subsystem.

The HFTQDEV **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFTQDEV, &arg)
int fildes;
struct hftqdresp arg;
```

The HFTQDEV operation stores information in the **hftqdresp** structure, which is described in detail in the **sys/hft.h** header file. The fields in the **hftqdresp** structure are described in **hft.h** File Structures for Query **ioctl** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Query Device (HFTQDEV) in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference*.

HFT Device Driver (HFTDD) User Interface Overview, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFTQSMGR ioctl Operation (Query Screen Manager)

Purpose

Queries the screen manager ring for each virtual terminal.

Description

The Query Screen Manager (HFTQSMGR) **ioctl** operation queries the screen manager ring for each virtual terminal, returning the virtual terminal ID and state of each. The file descriptor can be associated with any virtual terminal in the High Function Terminal (HFT).

The Query Screen manager **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFTQSMGR, &arg)
int fildes;
struct hfbuf arg;

struct hfbuf
{
        struct hftqstat *hf_bufp;
        long hf_buflen;
};
```

The **hf_bufp** field points to an **hfqstat** structure, into which the **ioctl** operation loads the requested information. This information includes the number of virtual terminals and the attributes of each of these terminals. The **hfbuf** and **hfqstat** structures are defined in the **sys/hft.h** header file. Query Screen Manager (HFTQSMGR) in **hft.h** File Structures for Query **ioctl** Operations gives information on using this structure.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Query Screen Manager (HFTQSMGR) in **hft.h** File Structures for Query **ioctl** Operations in *Files Reference*.

Understanding the Screen Manager Ring, HFT Device Driver (HFTDD) User Interface Overview, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts*.

HFTSBREAK ioctl Operation (Set Break Map)

Purpose

Changes an individual virtual terminal's break map.

Description

The Set Break Map (HFTSBREAK) **ioctl** operation changes an individual virtual terminal's break map. This facility is operational in Monitor Mode (MOM) only.

The default is to break on nothing. If a break bit is set (that is, set to On or to a value of 1), keyboard input data is reported through the **read** operation, instead of being put on the ring buffer. The **SIGMSG** signal is sent to the application to indicate that input data is available.

If the break bit is reset, however (to Off or to a value of 0), then the input data is placed in the MOM input buffer ring. (This assumes that the ring was previously defined.) The application can then retrieve the input from this MOM buffer. If no buffer was defined, then all keystrokes are returned through the read. (That is, the break map is not used.) The break map is used only when the terminal is in untranslated keystroke mode.

The Set Break Map ioctl Structure

The HFTSBREAK operation is invoked with the following:

```
int ioctl(fildes, HFTSBREAK, &arg)
int fildes;
struct hfbuf arg;

struct hfbuf
{
    struct break_map *hf_bufp;
    long hf_buflen;
};

struct break_map
{
    ulong vtmbrk[16];
};
```

The **hfbuf** and **break_map** structures are defined in the **sys/hft.h** header file. Use of the fields in these structures is described in Set Echo (HFTSECHO) and Break Maps (HFTSBREAK) in **hft.h** File Structures for Special **ioctl** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Set Echo (HFTSECHO) and Break Maps (HFTSBREAK) in **hft.h** File Structures for Special **ioctl** Operations in *Files Reference.*

HFT Device Driver (HFTDD) User Interface Overview, Understanding Monitor Mode, Echo and Break Map Structure, HFT ioctl Operations in *Kernel Extensions and Device Support Programming Concepts.*

HFTSECHO ioctl Operation (Set Echo Map)

Purpose

Changes an individual virtual terminal's echo map.

Description

The Set Echo Map (HFSECHO) **ioctl** operation changes an individual virtual terminal's echo map. In most instances, echoing is done by the line discipline selected by the user. However, the echo map is provided to permit echoing by the High Function Terminal Subsystem (HFTSS) of specific characters.

Echoing Keyboard Input in HFT

The echo map of a virtual terminal is operational only in KSR (Keyboard Send-Receive) mode. The defaults for KSR mode are:

- The echo map is set to echo all characters and control sequences.
- Send-Receive Mode (SRM) is set so that the HFT subsystem does no echoing.

The application must turn off the bits that the HFT subsystem should not echo. These bits usually include the escape character and the characters defined for interrupt, quit, and erase. The characters defined for kill, End of File (EOF), End of Line (EOL), pacing, and other characters can also be included. For the HFT to perform any echoing itself, SRM must be reset.

Invoking the Set Echo Operation

The HFTSECHO **ioctl** operation is invoked as follows:

```
int ioctl(fildes, HFTSECHO, &arg)
int fildes;
struct hfbuf arg;

struct hfbuf
{
    struct echo_map *hf_bufp;
    long hf_buflen;
};

struct echo_map
{
    ulong vtmecho[16];       /* 512–bit echo mask  */

};
```

The **hfbuf** and **echo_map** structures are defined in the **sys/hft.h** header file. The use of the fields in these structures is described in Set Echo (HFTSECHO) and Break Maps (HFTSBREAK) in **hft.h** File Structures for Special **ioctl** Operations.

The structure of the data in the echo map is described in Echo and Break Map Structure.

The Reconfigure (HRFCONF) **ioctl** operation can also be used to change the system default echo map.

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

The Reconfigure (HFRCONF) **ioctl** Operation.

Set Echo (HFTSECHO) and Break Maps (HFTSBREAK) in **hft.h** File Structures for Special **ioctl** Operations in *Files Reference.*

HFT Device Driver (HFTDD) User Interface Overview, HFT ioctl Operations, Understanding Echo Maps, Understanding Keyboard Send-Receive (KSR) Mode in *Kernel Extensions and Device Support Programming Concepts.*

Input Device Report read Operation

Purpose

Reports input data from the mouse, tablet, LPFKs, or valuator dials.

Description

The Input Device Report **read** operation reports input data from the mouse, tablet, lighted programmable function keys (LPFKs), or valuator dials.

The Input Device Report **read** operation is invoked by:

```
int read(fildes, buffer, size of (buffer))
int              fildes;
struct hflocator buffer;

struct hflocator
{
    char hf_esc;
    char hf_lbr;
    char hf_why;
    char hf_deltax[2];
    char hf_deltay[2];
    char hf_seconds[3];
    char hf_sixtyths;
    char hf_buttons;
    char hf_stype;
};
```

The **hflocator** structure is used to return input from the mouse, tablet, LPFKs, and valuator dial.

The **hflocator** structure is defined in the **sys/hft.h** header file. Its use is described in Input Device Report in **hft.h** File Structures for **read** Operations.

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Input Device Report in **hft.h** File Structures for **read** Operations in *Files Reference.*

Reading Input with the read Operation*Kernel Extensions and Device Support Programming Concepts.*

Untranslated Key Control read Operation

Purpose

Returns untranslated character data entered from the keyboard of an HFT device.

Description

The **read** operation can be used to request untranslated character data input from the keyboard. Requesting untranslated input requires that the HFXLATKBD bit in the **hfprotocol.hf_select** field be turned off. The Set Protocol Modes **write** Operation gives more information on setting this field.

The Untranslated Key Control **read** operation is invoked by:

```
int read(fildes, buffer, size of (buffer))
int              fildes;
struct hfunxlate buffer;

struct hfunxlate
{
    char hf_esc;
    char hf_lbr;
    char hf_ww;
    char hf_keypos;
    char hf_scancode;
    char hf_status[2];
    char hf_seconds[3];
    char hf_sixtyths;
};
```

The **hfunxlate** structure is defined in the **sys/hft.h** header file. For information on using the **hfunxlate** structure, see Untranslated Keyboard Input in **hft.h** File Structures for **read** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Untranslated Keyboard Input in **hft.h** File Structures for **read** Operations in *Files Reference*.

Set Protocol Modes **write** Operation.

Reading Input with the read Operation in *Kernel Extensions and Device Support Programming Concepts*.

Cancel Sound write Operation

Purpose

Removes sound requests from the speaker device buffer.

Description

The Cancel Sound **write** operation removes sound requests from the speaker device buffer. This **write** operation cancels all requests belonging to the process making the call. Only sound requests whose **HFEXECALWAYS** flag is set (when the **hfsound** control was initially sent with the Send Sound **write** Operation) are left in the speaker device.

An inactive terminal ignores the Cancel Sound **write** operation. Note that if a sound is currently being played, it cannot be terminated.

The Cancel Sound **write** operation is invoked by:

```
int write(fildes, buffer, buflen);
int fildes;
struct hfcansnd *buffer;
int buflen;

struct hfcansnd
{
        char hf_intro[HFINTROSZ];
};
```

The fields in the **hfcansnd** structure are defined in the **sys/hft.h** header file. Their use is described in Cancel Sound in **hft.h** File Structures for General **write** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Cancel Sound in **hft.h** File Structures for General **write** Operations in *Files Reference*.

Sound Device (Speaker) Interface.

Send Sound **write** Operation.

Understanding HFT Output write Operations, Sound Device (Speaker) Interface in *Kernel Extensions and Device Support Programming Concepts*.

Change Font Palette write Operation

Purpose

Redefining a virtual terminal's font palette.

Description

You can redefine a virtual terminal's font palette with the Change Font Palette **write** operation in the following manner:

```
int write(fildes, buffer, buflen);
int fildes;
struct hffont *buffer;
int buflen;

struct hffont {
  char hf_intro [HFINROSZ]
  char hf_sublen;
  char hf_subtype;
  char hf_primary [2];
  char hf_alt1 [2];
  char hf_alt2 [2];
  char hf_alt3 [2];
  char hf_alt4 [2];
  char hf_alt5 [2];
  char hf_alt6 [2];
  char hf_alt7 [2];
}
```

The fields in the **hffont** structure are defined in the **sys/hft.h** file. Their use is described in Change Fonts in **hft.h** File Structures for KSR **write** Operations.

If the font palette is changed, the data currently in the presentation space is lost. The cursor also reverts to the double underscore and is placed at the home position (first column, first row).

When the Change Font request is accepted, the presentation space size is adjusted. It is adjusted to the number of rows and columns that fit on the physical display screen for the new font size. All fonts in the font palette must be the same size.

When first opened, the virtual terminal chooses as the primary font the default font for the current display. All alternate font entries are also initialized to the default font. The default presentation space size for a new virtual terminal is 80 columns by 25 rows.

Note: The **hffont** structure contains, among other fields, several variables for storing alternate font attributes. The entire structure must be completed with valid font IDs to prevent failure of the Change Font Palette **write** operation. However, it is acceptable to repeat entries such as the primary font.

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Change Fonts in **hft.h** File Structures for KSR **write** Operations in *Files Reference.*

Understanding HFT Output write Operations in *Kernel Extensions and Device Support Programming Concepts.*

Change Physical Display write Operation

Purpose

Changes the physical display used by a virtual terminal.

Description

The Change Physical Display **write** operation changes the physical display to which the virtual terminal is logically attached. The user has the option of requesting a change to the default display or specifying a particular display. Physical display identifiers can be found by calling the Query Physical Device **ioctl** Operation.

The Change Physical Display **write** operation can be invoked by:

```
int write(fildes, buffer, buflen);
int fildes;
struct hfchgdsp *buffer;
int buflen;

struct hfchgdsp
{
        char hf_intro[HFINTROSZ];
        char hf_sublen;
        char hf_subtype;
        char hf_mode[2];
        char hf_rsvd1[8];
        char hf_devid[4];
        char hf_rsvd2[10];
};
```

The fields in the **hfchgdsp** structure are defined in the **sys/hft.h** header file. Their use is described in Change Physical Display in hft.h File Structures for General write Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

The Query Physical Device **ioctl** Operation.

Change Physical Display in **hft.h** File Structures for General **write** Operations in *Files Reference.*

Understanding HFT Output write Operations in *Kernel Extensions and Device Support Programming Concepts.*

Redefine Cursor Representation write Operation

Purpose

Redefining a virtual terminal's cursor representation.

Description

You can use the Redefine Cursor Representation **write** operation to redefine a virtual terminal's cursor representation. The cursor can have six different representations:

- Single underscore
- Double underscore
- Lower-half illuminated character cell
- Double mid-character line
- Fully illuminated character cell
- No cursor.

The Redefine Cursor Representation **write** operation is invoked by:

```
int write(fildes, buffer, buflen);
int fildes;
struct hfcursor *buffer;
int buflen;
```

The **hfcursor** structure is used for this request:

```
struct hfcursor
{
        char hf_intro[HFINTROSZ];
        char hf_sublen;
        char hf_subtype;
        char hf_rsvd;
        char hf_shape;
};
```

The fields in the **hfcursor** structure are defined in the **sys/hft.h** header file. Their use is described in Redefine Cursor Representation in **hft.h** File Structures for KSR **write** Operations.

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Redefine Cursor Representation in **hft.h** File Structures for KSR **write** Operations in *Files Reference*.

Understanding HFT Output write Operations in *Kernel Extensions and Device Support Programming Concepts*.

Screen Release write Operation

Description

The Screen Release **write** operation informs the operating system that the state of the display hardware can be changed. When a virtual terminal in Monitor mode is to be deactivated, its controlling process or group of processes receives a **SIGRETRACT** signal. This gives the program a chance to save the state of the display hardware, such as registers and refresh memory. After this is done, the program should use this operation to write to the terminal a screen release control.

Note that if the HFT does not receive the Screen Release request within 30 seconds of sending the **SIGRETRACT** signal, all processes in that virtual terminal group are sent a **SIGKILL** signal.

The Screen Release **write** operation can be invoked by:

```
int write(fildes, buffer, buflen);
int fildes;
struct hfmomscrel *buffer;
int buflen;

struct hfmomscrel
{
   char hf_intro[HFINTROSZ];
};
```

The **hfmomscrel** structure is defined in the **sys/hft.h** header file. Further information about the Screen Release **write** operation can be found in Screen Release in **hft.h** File Structures for MOM **write** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Screen Release in **hft.h** File Structures for MOM **write** Operations in *Files Reference.*

Understanding Monitor (MOM) Mode in *Kernel Extensions and Device Support Programming Concepts.*

Screen Request write Operation

Description

The Screen Request **write** operation permits a user program to perform direct operations on display hardware. A user program must first request permission to do so from the operating system, whether the user has already entered MOM mode. A **SIGGRANT** signal is sent to the application when the screen is granted.

The Screen Request **write** operation can be invoked by:

```
int write(fildes, buffer, buflen);
int fildes;
struct hfmomscreq *buffer;
int buflen;

struct hfmomscreq
{
        char hf_intro[HFINTROSZ];
        char hf_sublen;
        char hf_subtype;
        char hf_ringlen[2];
        char hf_ringoffset[4];
};
```

When requesting a screen, the user also has the option of requesting the use of a ring buffer. A ring buffer is a circular buffer into which device input can be placed, if the user so desires. This allows the application to avoid making **read** calls for input.

If this capability is desired, the **hf_ringlen** field should specify the size of the input ring buffer, as defined by the **hfmomring** structure. In this case, the program can access input placed directly in the buffer by the keyboard, mouse, tablet, LPFKs, or dials. The **read** operation is not needed for this access. Reading Input Data from a Ring Buffer provides guidance on extracting data from this buffer.

The **hfmomscreq** and **hfmomring** structures are defined in the **sys/hft.h** header file. Their use is described in Screen Request in **hft.h** File Structures for MOM **write** Operations.

If the user does not need to use a ring buffer, then the **hf_len** field of the **hf_intro** substructure should be set to the size of the introducer alone. In this case, input can be read with the standard **read** operation.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Screen Request in **hft.h** File Structures for MOM **write** Operations in *Files Reference*.

Understanding Monitor Mode (MOM), Understanding HFT Output write Operations, Requesting Screen Control in *Kernel Extensions and Device Support Programming Concepts*.

Send Sound write Operation

Purpose

Sends output to the speaker.

Description

The Send Sound **write** operation sends output to the speaker.

The Send Sound **write** operation is invoked by:

```
int write(fildes, buffer, buflen);
int fildes;
struct hfsound *buffer;
int buflen;

struct hfsound
{
        char hf_intro[HFINTROSZ];
        char hf_sublen;
        char hf_subtype;
        char hf_mode;
        char hf_rsvd;
        char hf_dur[2];
        char hf_freq[2];
};
```

The mode byte in the **hfsound** structure determines two aspects of the speaker sound:

- The sound command is executed whether the virtual terminal is active or not.
- The application is notified after the sound command executes.

Values for the duration of the sound and the frequency range are not checked. However, the valid frequency range is from 23 to 12000 hertz. There is no specified range for sound duration.

The fields in the **hfsound** structure are defined in the **sys/hft.h** header file. Their use is described in Write Sound in **hft.h** File Structures for General **write** Operations.

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Write Sound in **hft.h** File Structures for General **write** Operations in *Files Reference.*

Cancel Sound **write** Operation.

Understanding the HFT Output write Operations, Sound Device (Speaker) Interface in *Kernel Extensions and Device Support Programming Concepts.*

Set Dial Granularities write Operation

Purpose

Modifies the dial granularities.

Description

The Set Dial Granularities **write** operation allows the user program to set the dial granularities of any dial to a value from 2 to 8, inclusive.

The Set Dial Granularities **write** operation is invoked by:

```
int write(fildes, buffer, buflen);
int fildes;
struct hfdial_lpfk *buffer;
int buflen;
```

The fields in the **hfdial_lpfk** structure are defined in the **sys/hft.h** header file. Their use is described in Set LPFKs and Set Dial Granularities in **hft.h** File Structures for General **write** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Set LPFKs and Set Dial Granularities in **hft.h** File Structures for General **write** Operations in *Files Reference*.

Dials Interface, Understanding HFT Output write Operations in *Kernel Extensions and Device Support Programming Concepts*.

Set Keyboard LEDs write Operation

Purpose

Modifies the keyboard LEDs.

Description

The Set Keyboard LEDs **write** operation allows the user to set the Num Lock, Caps Lock, and Scroll Lock LEDs on or off.

The Set Keyboard LEDs **write** operation is invoked by:

```
int write(fildes, buffer, buflen);
int fildes;
struct hfkled *buffer;
int buflen;
```

The Set Keyboard LEDs **write** operation uses the **hfkled** structure. It contains the following fields:

```
struct hfkled
{
        char hf_intro[HFINTROSZ];
        char hf_sublen;
        char hf_subtype;
        char hf_ledselect;
        char hf_ledvalue;
};
```

The fields in the **hfkled** structure are defined in the **sys/hft.h** header file. Their use is described in Set Keyboard LEDs in **hft.h** File Structures for General **write** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Set Keyboard LEDs in **hft.h** File Structures for General **write** Operations in *Files Reference.*

Understanding HFT Output write Operations, Keyboard Hardware Reference in *Kernel Extensions and Device Support Programming Concepts.*

Set KSR Color Palette write Operation

Purpose

Specifies which colors to associate with a specified virtual terminal.

Description

The Set KSR Color Palette **write** operation specifies the color palette to be associated with a specified virtual terminal. The palette data in the color palette is specific to display adapters.

You can use the Set KSR Color Palette **write** operation to define a new color palette:

```
int write(fildes, buffer, buflen);
int fildes;
struct hfcolorpal *buffer;
int buflen;

struct hfcolorpal
{
        char hf_intro[HFINTROSZ];
        char hf_sublen;
        char hf_subtype;
        char hf_numcolor[2];
        char hf_palet[4][16];
};
```

The fields in the **hfcolorpal** structure are defined in the **sys/hft.h** header file. Their use is described in Set KSR Color Map in **hft.h** File Structures for KSR **write** Operations.

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Set KSR Color Map in **hft.h** File Structures for KSR **write** Operations in *Files Reference.*

Understanding HFT Output write Operations*Kernel Extensions and Device Support Programming Concepts.*

Set LPFKs write Operation

Purpose

Turns lights on the LPFKs on or off.

Description

The Set LPFKs **write** operation enables or disables the lighted programmable function keys.

The Set LPFKs **write** operation is invoked with:

```
int write(fildes, buffer, buflen);
int fildes;
struct hfdial_lpfk *buffer;
int buflen;
```

The fields in the **hfdial_lpfk** structure are defined in the **sys/hft.h** header file. Their use is described in Set LPFKs and Set Dial Granularities in **hft.h** File Structures for General **write** Operations.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Set LPFKs and Set Dial Granularities in **hft.h** File Structures for General **write** Operations in *Files Reference*.

Understanding HFT Output write Operations, Lighted Programmable Function Keys (LPFKs) Interface in *Kernel Extensions and Device Support Programming Concepts*.

Set Protocol Modes write Operation

Purpose

Modifies the protocol modes of a virtual terminal.

Description

Virtual terminal protocol modes determine how the virtual terminal interprets, translates, and returns data. The Set Protocol Modes **write** operation allows you to set these protocol modes by using the **write** Operation.

The Set Protocol Modes **write** operation is invoked by:

```
int write (fildes, buffer, buflen);
int  fildes;
struct hfprotcol *buffer;
int buflen;
```

The **hfprotocol** structure gives the protocol definitions:

```
struct hfprotocol
{
        char hf_intro[HFINTROSZ];
        char hf_sublen;
        char hf_subtype;
        char hf_select[2];
        char hf_value[2];
};
```

When issuing the Set Protocol Modes **write** operation, specify in the **hf_intro.hf_typehi** and **hf_intro.hf_typelo** fields one of the following two types:

- HFKSRPROH, HFKSRPROL
- HFMOMPROH, HFMOMPROL.

The selected type depends on whether you are sending the Set Protocol Modes **write** operation from within Keyboard Send-Receive (KSR) mode or Monitor (MOM) mode. Only certain protocol modes are valid in each of these modes. An attempt to set an invalid protocol mode results in rejection of the entire request.

The **hfprotocol** structure is defined in the **sys/hft.h** header file. For information on using the **hfprotocol** structure, see Set Protocol Modes in **hft.h** File Structures for General **write** Operations. Understanding Protocol Modes provides a list of possible protocol modes and how the corresponding bits in the **hfprotocol** structure should be set.

File

/usr/include/sys/hft.h

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Set Protocol Modes in **hft.h** File Structures for General **write** Operations in *Files Reference.*

Understanding HFT Output write Operations, Understanding Protocol Modes in *Kernel Extensions and Device Support Programming Concepts.*

Accented Characters

Description

Here are the valid sets of characters for each of the diacritics that the High Function Terminal (HFT) subsystem uses to validate the two-key nonspacing character sequence.

List of Diacritics Supported by the HFT Subsystem

There are seven diacritic characters for which sets of characters are provided:

- Acute
- Grave
- Circumflex
- Umlaut
- Tilde
- Overcircle
- Cedilla.

Valid Sets of Characters (Categorized by Diacritics)

Acute

Function	Code Value
Acute Accent	0xef
Apostrophe (Acute)	0x27
e Acute Small	0x82
e Acute Capital	0x90
a Acute Small	0xa0
i Acute Small	0xa1
o Acute Small	0xa2
u Acute Small	0xa3
a Acute Capital	0xb5
i Acute Capital	0xd6
y Acute Small	0xec
y Acute Capital	0xed
o Acute Capital	0xe0
u Acute Capital	0xe9

Grave

Function	**Code Value**
Grave Accent	0x60
a Grave Small	0x85
e Grave Small	0x8a
i Grave Small	0x8d
o Grave Small	0x95
u Grave Small	0x97
a Grave Capital	0xb7
e Grave Capital	0xd4
i Grave Capital	0xde
o Grave Capital	0xe3
u Grave Capital	0xeb

Accented Characters

Circumflex

Function	Code Value
^ Circumflex Accent	0x5e
a Circumflex Small	0x83
e Circumflex Small	0x88
i Circumflex Small	0x8c
o Circumflex Small	0x93
u Circumflex Small	0x96
a Circumflex Capital	0xb6
e Circumflex Capital	0xd2
i Circumflex Capital	0xd7
o Circumflex Capital	0xe2
u Circumflex Capital	0xea

Umlaut

Function	Code Value
Umlaut Accent	0xf9
u Umlaut Capital	0x9a
u Umlaut Small	0x81
a Umlaut Small	0x84
e Umlaut Small	0x89
i Umlaut Small	0x8b
a Umlaut Capital	0x8e
O Umlaut Capital	0x99
u Umlaut Capital	0x99
e Umlaut Capital	0xd3
i Umlaut Capital	0xd8

Tilde

Function	Code Value
Tilde Accent	0x7e
n Tilde Small	0xa4
n Tilde Capital	0xa5
a Tilde Small	0xc6
a Tilde Capital	0xc7
o Tilde Small	0xe4
o Tilde Capital	0xe5

Overcircle

Function	Code Value
Overcircle Accent	0x7d
a Overcircle Small	0x86
a Overcircle Capital	0x8f

Cedilla

Function	Code Value
Cedilla Accent	0xf7
c Cedilla Capital	0x80
c Cedilla Small	0x87

Related Information

Nonspacing Characters Overview, Keyboard Introduction in *Kernel Extensions and Device Support Programming Concepts*.

Echo and Break Map Structure

Description

The **echo_map** and **break_map** structures consist of 16 consecutive words of storage aligned on a word boundary. The bits are numbered as follows:

Word Number	Bits Contained by That Word
Word 0	Bits 0 – 31
Word 1	Bits 32 – 63
Word 2	Bits 64 – 95
Word 3	Bits 96 – 127
Word 4	Bits 128 – 159
Word 5	Bits 160 – 191
Word 6	Bits 192 – 223
Word 7	Bits 224 – 255
Word 8	Bits 256 – 287
Word 9	Bits 288 – 319
Word 10	Bits 320 – 351
Word 11	Bits 352 – 383
Word 12	Bits 384 – 415
Word 13	Bits 416 – 447
Word 14	Bits 448 – 479
Word 15	Bits 480 – 511.

All 512 bits of each map have meaning.

Bits 32 (0x20) through 255 (0xFF) are tested to see if the corresponding code point should be processed. For example, the code point 0x20 (32 decimal) is tested against the bit (33) in the map.

Mapped Multibyte Controls

Bit positions 256 (0x100) through 512 (0x200) are mapped as follows:

Bit Position	Mnemonic	Function
256 (0x100)	CBT	Cursor back tab
257 (0x101)	CHA	Cursor horizontal absolute
258 (0x102)	CHT	Cursor horizontal tab
259 (0x103)	CTC	Cursor tab stop control
260 (0x104)	CNL	Cursor next line
261 (0x105)	CPL	Cursor preceding line
262 (0x106)	CPR	Cursor position report
263 (0x107)	CUB	Cursor backward
264 (0x108)	CUD	Cursor down
265 (0x109)	CUF	Cursor forward
266 (0x10A)	CUP	Cursor position
267 (0x10B)	CUU	Cursor up
268 (0x10C)	CVT	Cursor vertical tab
269 (0x10D)	DCH	Delete character
270 (0x10E)	DL	Delete line
271 (0x10F)	DSR	Device status report
272 (0x110)	DMI	Disable manual input
273 (0x111)	EMI	Enable manual input
274 (0x112)	EA	Erase area
275 (0x113)	ED	Erase display
276 (0x114)	EF	Eras field

Bit Position	Mnemonic	Function
277 (0x115)	EL	Erase line
278 (0x116)	ECH	Erase character
279 (0x117)	GSM	
280 (0x118)	HTS	Horizontal tab stop
281 (0x119)	HVP	Horizontal and vertical position
282 (0x11A)	ICH	Insert character
283 (0x11B)	IL	Insert line
284 (0x11C)	IND	Index
285 (0x11D)	NEL	Next line
286 (0x11E)	PFK	PF key report
287 (0x11F)	INV	Invalid
288 (0x120)	INV	Invalid
289 (0x121)	RI	Reverse index
290 (0x122)	INV	Invalid
291 (0x123)	RIS	Reset to initial state
292 (0x124)	RM	Reset mode
293 (0x125)	SD	Scroll down
294 (0x126)	SL	Scroll left
295 (0x127)	SR	Scroll right
296 (0x128)	SU	Scroll up
297 (0x129)	SGR	Set graphic rendition
298 (0x12A)	SG0	Set G0 character set
299 (0x12B)	SG1	Set G1 character set
300 (0x12C)	SM	Set mode
301 (0x12D)	TBC	Tabulation clear
302 (0x12E)	VTS	Vertical tab stop
303 (0x12F)	SCP	Save cursor position
304 (0x130)	RCP	Restore cursor position
305 (0x131)	KSI	Keyboard status information
306 (0x132)	INV	Invalid or unsupported
511 (0x1FF)	INV	Invalid or unsupported.

Unmapped Data Stream Multibyte Controls

The following data stream multibyte controls are not mapped:

VTA	Virtual terminal addressability
VTD	Virtual terminal data
VTL	Virtual terminal device input
VTR	Virtual terminal raw keyboard input.

Related Information

Set Echo Map (HFTSECHO) **ioctl** Operation.

HFT ioctl Operations, Understanding Echo Maps in *Kernel Extensions and Device Support Programming Concepts*.

Requesting Screen Control and Specifying an Input Ring Buffer

Description

Although the virtual terminal is in Monitor mode, the program can perform direct operations on the display hardware only when granted permission by the operating system. The program first writes a screen request control using this call:

```
int write(fildes, buffer);
int fildes;
struct hfmomscreq *buffer;
```

A **SIGGRANT** signal is sent to the application when the screen is granted. If you do not want to specify or use a ring buffer, then set the **hf_len** field of the **hf_intro** structure to the size of the introducer only. In this case, read input with the standard read operation.

The hfmomscreq Structure and Fields

The screen control request uses the **hfmomscreq** structure:

```
struct hfmomscreq
{
        char hf_intro [HFINTROSZ];
        char hf_sublen;
        char hf_subtype;
        char hf_ringlen[2];
        char hf_ringoffset[4];
};
```

The **hfmomscreq** structure contains the following fields:

Field	**Value**
hf_intro.hf_len	Indicates the length of the request to the input ring buffer.
hf_intor.hf_typehi	HFMOMREQH
hf_intro.hf_typelo	HFMOMREQL
hf_sublen	2
hf_subtype	0
hf_ringlen[2]	Indicates the length of the ring in bytes.
hf_ringoffset[4]	Indicates the offset to the input buffer ring from the beginning of the **hf_ringlen** field.

If the **hf_ringlen** field specifies the size of the **hfmomring** structure, the program can access input (placed by the keyboard, mouse, tablet, LPFKs, and dials) directly into the buffer without issuing read calls.

The input ring buffer structure (also called the **hfmomring** structure) can be at any location in memory aligned on a word boundary. The **hf_ringoffset** field represents the difference between the ring buffer address and the address of the **hf_ringlen** field. It must be a positive value. Usually, the **hfmomring** ring buffer structure is defined so that it immediately follows the **hfmomscreq** structure in memory. Note that the compiler can implicitly insert one or more filler bytes between the two structures to align them at a memory address boundary. The value of the **hf_ringoffset** field must reflect such filler bytes.

The hfmomring Structure and Fields

Here is the **hfmomring** structure:

```
struct hfmomring
{
        char hf_rsvd[2];
        char hf_intreq;
        char hf_ovflow;
        unsigned hf_source;
        unsigned hf_sink
        int hf_unused[5];
        char hf_rdata[HFRDATA];
};
```

The **hfmomring** structure contains the following fields:

Field	**Value**
hf_intreq	Interrupt request can be set to 0xFF by the application.This causes the HFT subsystem to send a signal each time an input event occurs. Data entry into the input ring generates a **SIGMSG** signal. If this flag is set to the 0 default setting, then a signal is sent to the application only when the buffer goes from being empty to nonempty. The virtual terminal automatically resets this byte to 0 each time the virtual terminal stores input data into the ring buffer.
hf_ovflow	Determines whether the input buffer ring can accommodate more input information. A value of 0xFF indicates an overflow. A value of 0x00 indicates normal operation.
hf_source	Indicates the offset into the ring where the HFT puts data received. This offset starts from the beginning of the ring, so the minimum value for the virtual terminal offset is 32 bytes. After initializing this field at definition time, application programs must not alter this field. If a program attempts to alter it once it is initialized, continuation of software function cannot be assured.
hf_sink	The offset into the **hfmomring** structure from which the application reads data. This offset also starts from the beginning of the input ring, so the minimum value for this offset is 32 bytes. The application must modify this field whenever input data is removed from the ring.

Related Information

Screen Request **write** Operation.

Requesting Screen Release.

Understanding Monitor Mode (MOM) in *Kernel Extensions and Device Support Programming Concepts.*

Requesting Screen Release

Description

If a virtual terminal in Monitor mode is active, pressing the Next Window key causes a **SIGRETRACT** signal to be sent to the process or group of processes controlling that active virtual terminal. Before activating the next virtual terminal, the operating system allows the program to save the state of the display hardware, such as registers and refresh memory. The program should next send a Screen Release Control **write** operation to the terminal to inform the operating system that the state of the display hardware can be changed.

```
int write(fildes, buffer, size of (buffer));
int fildes;
struct hfmomscrel *buffer;
```

The screen release control is given by the **hfmomscrel** structure:

```
struct hfmomscrel
{
        char hf_intro [HFINTROSZ];
};
```

Field	Value
hf_intro.hf_len	Specifies the structure's length, including the ring buffer, minus 3.
hf_intro.hf_typehi	HFMOMRELH
hf_intro.hf_typelo	HFMOMRELL

After the display is released, the next virtual terminal is activated. If this is not done within 30 seconds of the receipt of the **SIGRETRACT** signal, all processes in that terminal group receive a **SIGKILL** signal. This safeguard prevents disabled programs from disrupting the next window function.

The program can issue a **pause** call if there is no work to do while the display is not available. When the Monitor mode virtual terminal is activated again with the Next Window key, the program receives a **SIGGRANT** signal. In other words, the program can resume direct output to the display. Do not assume that the display hardware state is the same as when the program released it.

Related Information

Screen Release **write** Operation.

Requesting Screen Control.

Understanding Monitor Mode (MOM) in *Kernel Extensions and Device Support Programming Concepts.*

Valid Multibyte Control Codes for Clearing and Setting Tab Controls

Description

There are four valid multibyte control codes for clearing and setting tab controls.

List of Valid Multibyte Control Codes

Mnemonic	**Function**
CTC	Cursor Tab Stop Control
HTS	Horizontal Tab Stop
TBC	Tabulation Clear
VTS	Vertical Tab Stop.

Description of Valid Multibyte Control Codes

CTC ESC [*PS* W Cursor Tab Stop Control

0	Sets a horizontal tab at the cursor.
1	Sets a vertical tab at the cursor.
2	Clears a horizontal tab at the cursor.
3	Clears a vertical tab at the cursor.
4	Clears all horizontal tabs on the line.
5	Clears all horizontal tabs.
6	Clears all vertical tabs.

Sets or clears one or more tabulation stops according to the specified parameter. Tab stops on the first or last column cannot be cleared. When horizontal tab stops are set or cleared, the number of lines affected is all (if Tabulation Stop Mode is set) or one (if TSM is reset). This control does not change the position of characters already in the presentation space.

HTS ESC H Horizontal Tab Stop

Sets a horizontal tab stop at the current horizontal position. If Tabulation Stop Mode is set, then the tab stop applies only to this line. If TSM is reset, then the tab stop applies to all PS lines. This control does not change the positioning of characters already in the presentation space.

TBC ESC [*PS* g Tabulation Clear

0	Horizontal tab at the cursor column
1	Vertical tab at line indicated by the cursor
2	Horizontal tabs on the line
3	Horizontal tabs in the presentation space
4	Vertical tabs in the presentation space.

Clears tabulation stops specified by the parameters. Horizontal tab changes affect only the line indicated by the cursor if Tabulation Stop Mode is set. Horizontal tab changes affect all lines if TSM is reset. Any parameters not listed above are ignored. This control does not change the positioning of characters already in the presentation space.

Clearing and Setting Tab Controls

VTS	ESC I	Vertical Tab Stop

Sets a vertical tab stop at the line indicated by the cursor. This control does not change the positioning of characters already in the presentation space.

Related Information

Multibyte Controls in Data Stream Data Overview in *Kernel Extensions and Device Support Programming Concepts*.

Valid Multibyte Control Codes for Controlling Cursor Movement

Description

There are 18 valid multibyte control codes for controlling cursor movement.

List of Valid Multibyte Control Codes

Mnemonic	Function
CBT	Cursor Back Tab
CHA	Cursor Horizontal Absolute
CHT	Cursor Horizontal Tab
CNL	Cursor Next Line
CPL	Cursor Preceding Line
CPR	Cursor Position Report
CUB	Cursor Backward
CUD	Cursor Down
CUF	Cursor Forward
CUP	Cursor Position
CUU	Cursor Up
CVT	Cursor Vertical Tab
HVP	Horizontal and Vertical Position
IND	Index
NEL	Next Line
RCP	Restore Cursor Position
RI	Reverse Index
SCP	Save Cursor Position.

Description of Valid Multibyte Control Codes

CBT ESC [*PN* Z Cursor Back Tab

Moves the cursor back the number of horizontal tab stops specified by the *PN* variable. Tab stops are always set at the first and last columns of each line. If the cursor is already in the first column of a line and HFWRAP mode is set, the cursor moves to the last column. If AUTONL mode is also set, the cursor moves to the last column of the previous line. In this case, if the cursor is already on the first row of the presentation space, it moves to the last row.

CHA ESC [*PN* G Cursor Horizontal Absolute

Moves the cursor to the column specified by the *PN* variable, unless the column exceeds the PS width. If the column exceeds the PS width, the cursor moves to the PS column farthest to the right.

CHT ESC [*PN* I Cursor Horizontal Tab

Moves the cursor position forward to the tab stop specified by the *PN* variable. If the cursor is already in the last column of a line and HFWRAP mode is set, then the cursor returns to the first column of the line. If AUTONL mode is also set, then the cursor moves to the first column of the next line. In this case, if the cursor is already on the last line of the PS, then the cursor moves to the first column of the first line. The HT (horizontal tab) single–byte control does not cause wrapping to occur.

CNL ESC [*PN* E Cursor Next Line

Moves the cursor down the number of lines specified by the *PN* variable, and over to the first position of that line. If the cursor is already on the bottom PS line and HFWRAP mode is not set, it is positioned at the beginning of that line. If HFWRAP mode is set, the cursor wraps from the bottom line to the top PS line.

CPL ESC [*PN* F Cursor Preceding Line

Moves the cursor back the number of lines specified by the *PN* variable, and over to the first position of that line. If the cursor is already on the top PS line and HFWRAP mode is not set, the cursor is positioned at the beginning of that line. If HFWRAP mode is set, the cursor wraps from the top line to the bottom line of the PS.

CPR ESC [*PN* ; *PN* R Cursor Position Report

Reports the current cursor position. The first numeric parameter is the line number. The second parameter is the column. Line and column values are sent to the application as information. However, if the information is received by the virtual terminal, it is treated as a CUP control.

CUB ESC [*PN* D Cursor Backward

Moves the cursor backward on the line the specified number of columns. If this cursor movement exceeds the left PS boundary and HFWRAP mode is not set, the cursor stops at the leftmost PS position. If HFWRAP mode is set, the cursor wraps from the leftmost column to the rightmost column of the preceding PS line. In HFWRAP mode the cursor also wraps from the home to the rightmost bottom position of the PS.

CUD ESC [*PN* B Cursor Down

Moves the cursor down the number of lines specified by the *PN* variable. If this cursor movement exceeds the bottom PS boundary and HFWRAP mode is not set, the cursor stops on the last PS line. If HFWRAP mode is set, the cursor wraps from the bottom line to the top line of the PS.

CUF ESC [*PN* C Cursor Forward

Moves the cursor forward on the line the specified number of columns. If this cursor movement exceeds the right PS boundary and HFWRAP mode is not set, the cursor stops at the rightmost PS position. If HFWRAP mode is set, the cursor wraps from the rightmost column to the leftmost column of the following line in the PS. In HFWRAP mode, the cursor also wraps from rightmost bottom position to the home position of the PS.

CUP ESC [*PN* ; *PN* H Cursor Position

Moves the cursor to the line specified by the first parameter and to the column specified by the second parameter. If this movement crosses a PS boundary, the cursor stops at the PS boundary.

CUU ESC [*PN* A Cursor Up

Moves the cursor up the specified number of lines. If this cursor movement exceeds the top PS boundary and HFWRAP mode is not set, the cursor stops on the first PS line. If HFWRAP mode is set, the cursor wraps from the top line to the bottom line in the PS.

CVT ESC [*PN* Y Cursor Vertical Tab

Moves the cursor down the specified number of vertical tab stops. Tab stops are assumed at the top and bottom PS lines. If there are not enough vertical tab stops in the PS and if HFWRAP mode is not set, the cursor stops on the last line in the PS. If HFWRAP mode is set, the cursor wraps from the bottom line to the top line of the PS.

HVP ESC [*PN* ; *PN* f Horizontal and Vertical Position

Moves the cursor to the line specified by the first parameter and to the column specified by the second parameter. If this movement would cross a PS boundary, the cursor stops at the current PS boundary.

IND ESC D Index

Moves cursor down one line. If the cursor is already on the bottom line of the PS, then the top line is lost, the other lines move up one line, and a blank line becomes the new bottom line.

NEL ESC E Next Line

Moves the cursor to the first position of the following line. If the cursor is already on the bottom line of the PS, then the top line is lost, the other lines move up one, and a blank line becomes the new bottom line.

RI ESC L Reverse Index

Moves the cursor up one line, unless the cursor is already on the top PS line. In that case, if HFWRAP mode is not set, then the cursor does not move. If HFWRAP mode is set, the cursor moves to the bottom line of the PS. The cursor's column position does not change.

RCP ESC [u Restore Cursor Position

Moves the cursor to the position saved by the last SCP control. If no SCP has been received, then the cursor position is set to the first character of the first line. This is a private control that conforms to the ANSI standards for private controls. This control has no terminal function when received from the keyboard.

SCP ESC [s Save Cursor Position

Saves the current cursor position. Any previously saved cursor position is lost. The cursor can be restored to this new position with an RCP control. This is a private control that conforms to the ANSI standards for private controls. This control has no terminal function when received from the keyboard.

Related Information

Multibyte Controls in Data Stream Data Overview in *Kernel Extensions and Device Support Programming Concepts.*

Valid Multibyte Control Codes for Erasing Areas, Displays, Lines, and Fields

Description

There are four valid multibyte control codes for erasing areas, displays, lines, and fields.

List of Valid Multibyte Control Codes

Mnemonic	Function
EA	Erases to end of the area, or from start of area, or all of area.
ED	Erase to end of display, or from start of the display, or all of the display.
EF	Erase to end of the field, or from start of the field, or all of the field.
EL	Erase to end of the line, or from start of the line, or all of the line.

Description of Valid Multibyte Control Codes

EA	ESC [0 O	Erase to End of Area
	ESC [1 O	Erase from Start of Area
	ESC [2 O	Erase All of Area.

Erases a specified area. Erased characters are replaced with empty spaces. Erase to End of Area erases the area from the position indicated by the cursor through the end of the area. Erase from Start of Area erases from the start of the area until the position indicated by the cursor. Erase All of Area erases the entire area in which the cursor lies.

This control is treated like an EL control sequence.

ED	ESC [0 J	Erase to End of Display
	ESC [1 J	Erase from Start of Display
	ESC [2 J	Erase All of Display.

Erases certain characters within the presentation space. Erased characters are replaced with empty spaces. Erase to End of Display erases the character indicated by the cursor and all following characters in the PS. Erase from Start of Display erases the first character of first line and the following characters up to and including the character indicated by the cursor. Erase All of Display erases all the characters on the PS.

EF	ESC [0 N	Erase to End of Field
	ESC [1 N	Erase from Start of Field
	ESC [2 N	Erase All of Field.

Erases certain characters between horizontal tab stops. Erased characters are replaced with empty spaces. Erase to End of Field erases the character indicated by the cursor and all following characters before the next tab stop. Erase from Start of Field erases the character at the tab stop preceding the cursor and the following characters up to and including the character indicated by the cursor. Erase All of Field erases the character at the tab stop preceding the cursor and the following characters up to and including the character at the tab stop following the cursor. Tab stops are assumed at the first and last columns of the PS when executing this control.

EL	ESC [0 K	Erase to End of Line
	ESC [1 K	Erase from Start of Line
	ESC [2 K	Erase All of Line.

Erases certain characters within a line. Erased characters are replaced with empty spaces. Erase to End of Line erases the character indicated by the cursor and all following characters on the line. Erase from Start of Line erases the first character of first line and the following characters up to and including the character indicated by the cursor. Erase All of Line erases all the characters on the line.

Related Information

Multibyte Controls in Data Stream Data Overview in *Kernel Extensions and Device Support Programming Concepts.*

Valid Multibyte Control Codes for Inserting and Deleting Lines and Characters

Description

There are five valid multibyte control codes for inserting and deleting lines and characters.

List of Valid Multibyte Control Codes

Mnemonic	Function
DCH	Delete Character
DL	Delete Line
ECH	Erase Character
ICH	Insert Character
IL	Insert Line.

Description of Valid Multibyte Control Codes

DCH ESC [*PN* P Delete Character

Deletes the cursor character and the following characters specified by the *PN* variable minus 1 on the line indicated by the cursor. The characters following the deleted characters on the line overlay the deleted character positions. The line is cleared from the end of the line to the edge of the presentation space. If the number of characters to be deleted exceeds the number of columns from the cursor to the right boundary of the PS, then all the characters from the cursor to the PS boundary are replaced with empty spaces and a DSR control sequence identifying an error is returned to the application.

DL ESC [*PN* M Delete Line

Deletes the specified line and the lines that follow it (as specified by the *PN* variable minus 1) in the PS. The lines following the deleted lines are scrolled up the number of lines indicated by the *PN* variable and the number of blanks lines (specified by the *PN* variable) are placed at the bottom of the PS. If the area extending from the line indicated by the cursor to the bottom of the PS contains fewer lines than specified by the *PN* variable, the line indicated by the cursor and all the following PS lines are replaced with empty lines.

ECH ESC [*PN* X Erase Character

Erases the character indicated by the cursor and the characters that follow it (as specified by the *PN* variable minus 1) on that line. Erased characters are replaced with empty spaces. If the area between the cursor and PS right boundary contains fewer characters than specified by the *PN* variable, then the character indicated by the cursor and all the following characters on the line are replaced by empty spaces.

ICH ESC [*PN* @ Insert Character

Inserts the number of empty spaces specified by the *PN* variable before the character indicated by the cursor. The string of characters starting with the character indicated by the cursor and ending with last character of the line are shifted to the right the number of columns indicated by the *PN* variable. Characters shifted past the right boundary of the PS are lost. The cursor does not move.

IL	ESC [*PN* L	Insert Line

Inserts the number of empty lines specified by the *PN* variable before the line indicated by the cursor. The line indicated by the cursor is scrolled down. The cursor position on the screen is not affected.

Related Information

Multibyte Controls in Data Stream Data Overview in *Kernel Extensions and Device Support Programming Concepts.*

Valid Multibyte Control Codes for Performing Miscellaneous Tasks

Description

There are 15 valid multibyte control codes for performing miscellaneous tasks.

List of Valid Multibyte Control Codes

Mnemonic	Function
DSR	Device Status Report Request
DMI	Disable Manual Input
EMI	Enable Manual Input
KSI	Keyboard Status Information
PFK	PF Key Report
RIS	Reset to Initial State
RM	Reset Mode
SGR	Set Graphic Rendition
SG0A	Set G0 Character Set
SG1A	Set G1 Character Set
SM	Set Mode
VTA	Virtual Terminal Addressability
VTD	Virtual Terminal Data
VTL	Virtual Terminal Device Input
VTR	Virtual Terminal Raw Keyboard Input.

Description of Valid Multibyte Control Codes

DSR ESC [*PN* n Device Status Report Request

6	Request Cursor Position Report
13	Error Report.

A request cursor position report (CPR) sends a cursor position report from the virtual terminal to the application. An error report is sent from the virtual terminal to the application when the virtual terminal receives an invalid control sequence. Error reports are private reports that conform to the ANSI standard for private parameters.

DMI ESC " (Left Quote) Disable Manual Input

This control, when received in an output data stream, causes keyboard input to this terminal to be ignored. This control is ignored when received from the keyboard.

EMI ESC b Enable Manual Input

This control, when received in an output data stream, restarts keyboard input recognition and buffering if previously disabled with a DMI multibyte control. This control is ignored when received from the keyboard.

KSI ESC [*PS* p Keyboard Status Information

The virtual terminal generates this control whenever HFHOSTS and HFXLATKBD are set and the status of the keyboard changes. Each selective parameter is the character-coded decimal value of a keyboard status byte. For example, if the keyboard has two status bytes, the control sequence is ESC [*xxx*;*yyy* p, where *xxx* is the value of the high-order byte and *yyy* is the value of the low-order byte. This is a private control that conforms to the ANSI standards for private control sequences. The virtual terminal display handler ignores this sequence whether it is received from the application or echoed.

PFK	ESC [*PN* q	PF Key Report

The control sequence is sent by the virtual terminal to the application when a program function key (PFK) code is received from the keyboard. The parameter *PN* is a PF key number from 1 to 255. This is a private control that conforms to the ANSI standards for private control sequences. This sequence is ignored by the virtual terminal display handler whether received from the application or echoed.

RIS	ESC c	Reset to Initial State

Resets the virtual terminal to the state of a newly opened virtual terminal: erases all PS data, places the cursor at the home position, resets graphic rendition to normal, resets subscripting and superscripting, and sets tab stops, modes, keyboard map, character maps, and echo maps to their default values.

RM	ESC [PS l	Reset Mode
20	LNM	Line Feed New Line Mode
4	IRM	Insert Replace Mode
12	SRM	Send Receive Mode
18	TSM	Tabulation Stop Mode
?21	CNM	Carriage Return New Line Mode
?7	AUTONL	Wrap to following line when end of current line is reached.

Resets the modes specified in the parameter string. Multiple parameters must be separated by semicolons. The modes that can be reset are listed above with the appropriate parameter code. All other mode parameters are ignored.

If LNM is reset, the line feed moves the cursor position down one line.

If IRM is reset, a graphic character sent to the display is also placed at the cursor, but all symbols at and to the right of the cursor on the same line are shifted right one column position. Characters shifted from the last column on the line are discarded.

If SRM is reset and the echo map has been set correctly, then translated keyboard input is echoed by the virtual terminal.

If CNM is reset, the carriage return moves the cursor position to the first character of the line indicated by the cursor.

If AUTONL is reset, the cursor stays on the last column of the current line.

If TSM is reset, horizontal tabulation changes affect only the line indicated by the cursor.

SGR	ESC [*PS* m	Set Graphic Rendition
0	Normal (none of attributes 1 to 9)	
1	Bold or Bright	
4	Underscore	
5	Slow Blink	
7	Negative (reverse image)	
8	Canceled On (invisible: set to background color)	
10	Primary Font	
11	First Alternate Font	
12	Second Alternate Font	
13	Third Alternate Font	
14	Fourth Alternate Font	
15	Fifth Alternate Font	
16	Sixth Alternate Font	
17	Seventh Alternate Font	
30	Color palette entry 0 foreground	
31	Color palette entry 1 foreground	
32	Color palette entry 2 foreground	
33	Color palette entry 3 foreground	
34	Color palette entry 4 foreground	
35	Color palette entry 5 foreground	
36	Color palette entry 6 foreground	
37	Color palette entry 7 foreground	
40	Color palette entry 0 background	
41	Color palette entry 1 background	
42	Color palette entry 2 background	
43	Color palette entry 3 background	
44	Color palette entry 4 background	
45	Color palette entry 5 background	
46	Color palette entry 6 background	
47	Color palette entry 7 background	
90	Color palette entry 8 foreground	
91	Color palette entry 9 foreground	
92	Color palette entry 10 foreground	
93	Color palette entry 11 foreground	
94	Color palette entry 12 foreground	
95	Color palette entry 13 foreground	
96	Color palette entry 14 foreground	
97	Color palette entry 15 foreground	
100	Color palette entry 8 background	
101	Color palette entry 9 background	
102	Color palette entry 10 background	
103	Color palette entry 11 background	
104	Color palette entry 12 background	
105	Color palette entry 13 background	
106	Color palette entry 14 background	
107	Color palette entry 15 background.	

Causes the next characters received in the data stream or from the keyboard to have the attributes specified by the parameter string. Any parameter not listed above is ignored. Multiple parameters are processed in the order listed.

These parameters remain in effect for all following characters until they are reset. To reset one of these attributes, specify NORMAL and then reinstate the desired parameters.

The capabilities of the physical display used by the virtual terminal determine whether the characters actually have the requested attributes.

The attributes corresponding to parameters 1 through 9 are cumulative. For example, specifying UNDERSCORE and then specifying BLINK causes following characters to blink and be underscored.

Parameters 10 through 17 select a font from the font palette. For example, selecting 15 as a parameter changes the font being used by the display to the fifth alternate font in the font palette.

Parameters 30 through 37 and 90 through 97 select a foreground color from the color palette. Parameters 40 through 47 and 100 through 107 select a background color from the color palette.

SG0A	ESC (f	Set G0 Character Set
SG0B	ESC , f	Set G0 Character Set (Alternate form)
<	Display Symbols 32-255	

Designates the set of characters to use as the G0 set when the G0 set is invoked by SI.

SG1A	ESC) f	Set G1 Character Set
SG1B	ESC - f	Set G1 Character Set (Alternate)
<	Display Symbols 32-255	

Designates the set of characters to use as the G1 set when the G1 set is invoked by SO.

SM	ESC [*PS* h	Set Mode
20	LNM	Line Feed - New Line Mode (default = set)
4	IRM	Insert Replace Mode (default = reset)
12	SRM	Send Receive Mode (default = set)
18	TSM	Tabulation Stop Mode (default = reset)
?21	CNM	Carriage Return - New Line Mode (default = reset)
?7	AUTONL	Wrap to next line when end of line reached (default = set).

Sets the modes specified in the parameter string. Multiple parameters must be separated by semicolons. The modes that can be set are listed with the appropriate parameter code. All other mode parameters are ignored.

If LNM is set, the line feed moves the cursor position to the first position of the next line.

If IRM is set, a graphic character sent to the display is placed at the cursor, replacing the symbol already there.

If SRM is set, translated keyboard input is never echoed by the virtual terminal, but is immediately sent to the application.

If CNM is set, the carriage return causes the cursor to move to the first position of the next line.

If AUTONL is set, the cursor moves to the first column position of the following line.

If TSM is set, then horizontal tabulation changes affect all lines.

VTA	ESC [r	Virtual Terminal Addressability

This private control sequence precedes a binary header and associated data that provide status information on the IBM 5081 Display Adapter.

VTD	ESC [x	Virtual Terminal Data

This private control sequence precedes a binary header and associated data. The block of data can be in formats other than character-coded data, such as binary format.

VTL	ESC [y	Virtual Terminal Device Input

This private control sequence precedes binary format input data from a mouse, tablet, LPFK, or valuator device.

VTR	ESC [w	Virtual Terminal Raw Keyboard Input

This private control sequence precedes untranslated keyboard input data, which is in a binary format.

Related Information

Multibyte Controls in Data Stream Data Overview in *Kernel Extensions and Device Support Programming Concepts*.

Valid Multibyte Control Codes for Scrolling

Description

There are four valid multibyte control codes for scrolling.

List of Valid Multibyte Control Codes

Mnemonic	Function
SD	Scroll Down
SL	Scroll Left
SR	Scroll Right
SU	Scroll Up.

Description of Valid Multibyte Control Codes

SD ESC [*PN* T Scroll Down

Moves all the PS lines down the number of lines indicated by the *PN* variable. The bottom lines indicated by the *PN* variable are lost, and the empty lines indicated by the *PN* variable are put at the top of the presentation space. Physical cursor position does not change with the scroll.

SL ESC [*PN* SP @ Scroll Left

Moves to the left all the PS characters the number of column positions indicated by the *PN* variable. The characters in the leftmost PS columns specified by the *PN* variable are lost, and empty spaces are put in the rightmost columns of all lines specified by the *PN* variable. Physical cursor position does not change with the scroll.

SR ESC [*PN* SP A Scroll Right

Moves all the PS characters the number of column positions to the right indicated by the *PN* variable. The characters in the rightmost PS columns indicated by the *PN* variable are lost, and empty spaces are put in the leftmost columns specified by the PN variable. Physical cursor position does not change with the scroll.

SU ESC [*PN* S Scroll Up

Moves all the PS lines up the number of lines indicated by the *PN* variable. The number of top lines indicated by the *PN* variable are lost, and the number of empty lines indicated by the *PN* variable are put at the bottom of the presentation space. The physical cursor position does not change with the scroll.

Implementation Specifics

This routine is part of AIX Base Operating System (BOS) Runtime.

Related Information

Multibyte Controls in Data Stream Data Overview in *Kernel Extensions and Device Support Programming Concepts.*

Chapter 7. Logical Volume Subsystem

Physical Volumes and the Logical Volume Device Driver

In a discussion of how the logical volume device driver (LVDD) interacts with physical volumes the following topics are relevant:

- Direct access storage devices (DASDs)
- Physical volumes
  - Implementation limitations
  - Reserved Sectors
- The logical volume device driver structure
- Interface to physical disk device drivers
- Logical volumes and bad blocks.

Direct Access Storage Devices (DASDs)

Direct access storage devices (DASDs) are *fixed* or *removable* storage devices. Typically, these devices are (hard) disks. A fixed-storage device is any storage device defined by the person who administers your system during system configuration to be an integral part of the system DASDs. The AIX Base Operating System detects an error if a fixed-storage device is not available at some time during normal operation.

A removable storage device is any storage device defined by the person who administers your system during system configuration to be an optional part of the system DASD. The removable storage device can be removed from the system at any time during normal operation. As long as the device is logically unmounted first, the AIX operating system will *not* detect an error.

The following types of devices are *not* considered DASDs and are not supported by the logical volume manager (LVM):

- Diskettes
- CD-ROM (compact disk read-only memory)
- WORM (write-once read-mostly).

DASDs Device Block Level Introduction

The DASD *device block* (or *sector*) level is the level at which a processing unit may request low-level operations on a device block address basis. Typical low-level operations for DASD are `read-sector`, `write-sector`, `read-track`, `write-track`, and `format-track`.

A DASD stores data in a way that allows for its rapid retrieval from random addresses as a stream of one or more blocks. Many DASDs perform best when the blocks to be retrieved are close (in physical address) to each other.

DASDs consist of a set of flat, circular, rotating platters. Each platter has one or two sides on which data is stored. Platters are read by a set of nonrotating, but positionable, read or read/write *heads* that move together as a unit. The following are terms used when discussing DASD device block operations:

sector A contiguous, fixed-size block of data on a DASD. To maintain compatibility with the traditional UNIX(TM) model of DASD, every sector of every AIX DASD is defined to be exactly 512 bytes.

track A track is a contiguous set of sectors on a single DASD. A track corresponds to the surface area of a single platter swept out by a single head while the head remains stationary.

An AIX DASD contains at least 17 sectors per track. Otherwise, the number of sectors per track is not defined architecturally and is device-dependent. A typical AIX DASD track can contain 17, 35, or 75 sectors.

An AIX DASD might contain 1024 tracks. The number of tracks per DASD is not defined architecturally and is device-dependent.

head A head is a positionable entity that can read and write data from a given track located on one side of a platter. Usually a DASD has a small set of heads that move from track to track as a unit.

There must be at least 4 heads on a DASD. Otherwise, the number is not defined architecturally and is device-dependent. A typical DASD might have 8 heads.

cylinder The path swept out on the entire set of platters that can be read or written by the set of heads (when stationary). This path is called a cylinder. If a DASD has *n* number of vertically aligned heads, a cylinder is composed of *n* number of vertically aligned tracks.

Physical Volumes

A physical volume is a DASD structured for *physical level* requests. The physical level is the level at which a processing unit can request device-independent operations on a physical block address basis. A physical volume is composed of the following:

- A device-dependent reserved area
- A variable number of physical blocks that serve as DASD descriptors
- An integral number of partitions, each containing a fixed number of physical blocks.

When performing I/O at a physical level, no bad-block relocation is supported. Bad blocks are not hidden at this level as they are at the *logical level*. Typical operations at the physical level are `read-physical-block` and `write-physical-block`.

The following are terms used when discussing DASD volumes:

block A contiguous, 512-byte region of a physical volume that corresponds in size to a DASD sector.

partition A set of blocks (with sequential cylinder, head, and sector numbers) contained within a single physical volume.

The number of blocks in a partition as well as the number of partitions in a given physical volume are both fixed when the physical volume is installed in a volume group. Every physical volume in a volume group has exactly the same partition size. There is no restriction on the types of DASD devices (for example, SCSI, ESDI, or IPI) that may be placed in a given volume group.

Note: A given physical volume must be assigned to a volume group before that physical volume may be used by the AIX Base Operating System.

Physical Volume Implementation Limitations

When composing a physical volume from a DASD, the following implementation restrictions apply to DASD characteristics:

- 1 to 32 physical volumes per volume group.
- 1 to 1016 physical partitions per physical volume.
- The partition size is restricted to 2***n* bytes, for 20 <= *n* <= 28.
- The physical block size is restricted to 512 bytes.

Physical Volume Layout

A physical volume consists of a logically contiguous string of physical sectors. Sectors are numbered 0 through *LPSN*, where *LPSN* is the last physical sector number on the physical volume. The total number of physical sectors on a physical volume is *LPSN* + 1. The actual physical location and physical order of the sectors is transparent to the sector numbering scheme.

Note: Sector numbering applies to user-accessible data sectors only. Spare sectors and customer engineer (CE) sectors are not included. (CE sectors are reserved for use by diagnostic test routines or microcode.)

Reserved Sectors on a Physical Volume

A physical volume reserves the first 128 sectors to store various types of DASD configuration and operation information. The **<sys/hd_psn.h>** file describes the information stored on the reserved sectors. In this file, the locations of the items in the reserved area are expressed as physical sector numbers and the lengths of those items are in number of sectors.

The 128-sector reserved area of a physical volume includes a *boot record*, the *bad-block directory*, and the *LVM record*. The boot record consists of one sector containing information that allows the read-only system (ROS) to boot the system. A description of the boot record can be found in the **<sys/bootrecord.h>** file.

The boot record also contains the **pv_id** field. This field is a 64-bit number uniquely identifying a physical volume. This identifier is assigned by the manufacturer of the physical volume. However, if a physical volume is part of a volume group, the **pv_id** field may be assigned by the LVM.

The bad-block directory records the blocks on the physical volume that have been diagnosed as unusable. The structure of the bad-block directory and its entries can be found in the **<sys/bbdir.h>** file.

The LVM record consists of one sector and contains information used by the LVM when the physical volume is a member of the volume group. The LVM record is described in the **<lvmrec.h>** file.

Sectors Reserved for the Logical Volume Manager (LVM)

If a physical volume is part of a volume group, the physical volume is used by the LVM and contains two additional reserved areas. One contains the volume group descriptor area/volume group status area and follows the first 128 reserved sectors. The other is an area at the end of the physical volume reserved as a relocation pool for bad blocks that must be software-relocated. Both of these areas are described by the LVM record. The space between these last two reserved areas is divided into equal-sized partitions.

The volume group descriptor area (VGDA) is divided into the following:

- The volume group header

  This header contains general information about the volume group and a time stamp used to verify the consistency of the VGDA.

- A list of logical volume entries

  The logical volume entries describe the states and policies of logical volumes. This list defines the maximum number of logical volumes allowed in the volume group. The maximum is specified when a volume group is created.

- A list of physical volume entries

  The size of the physical volume list is variable because the number of entries in the partition map can vary for each physical volume. For example, a 200M byte physical volume with a partition size of 1M byte has 200 partition map entries.

- A name list

  This list contains the special file names of each logical volume in the volume group.

- A volume group trailer

  This trailer contains an ending time stamp for the volume group descriptor area.

  When a volume group is varied online, at least two readable copies of the volume group descriptor area are necessary in order to perform recovery operations. (The *vary-on* operation, performed by using the **varyonvg** command, makes a volume group available to the system.)

  A volume group with only one physical volume must contain two copies of the physical volume group descriptor area. For any volume group containing more than one physical volume, there are at least three on-disk copies of the volume group descriptor area. The default placement of these areas on the physical volume is as follows:

- For the first physical volume installed in a volume group, two copies of the volume group descriptor area are placed on the physical volume.

- For the second volume installed in a volume group, one copy of the volume group descriptor area is placed on the physical volume.

- For the third physical volume installed in a volume group, one copy of the volume group descriptor area is placed on the physical volume. The second copy is removed from the first volume.

- For additional physical volumes installed in a volume group, one copy of the volume group descriptor area is placed on the physical volume.

  When a vary-on operation is performed, a majority of all volumes containing a volume group descriptor area must be able to come online before the vary-on operation is considered successful. A majority ensures that at least one copy of the volume group descriptor areas used to perform recovery was also one of the volume group descriptor areas used during the previous *vary-off* operation. If this is not the case, the consistency of the volume group descriptor area cannot be insured.

The Logical Volume Device Driver

The Logical Volume Device Driver (LVDD) is a pseudo-device driver that operates on logical volumes through the **/dev/lv***n* special file. Like the physical disk device driver, this pseudo device driver provides character and block entry points with compatible arguments. Each volume group has an entry in the kernel *device switch table.* Each entry contains entry points for the device driver and a pointer to the volume group data structure. The logical volumes of a volume group are distinguished by their minor numbers.

Character I/O requests are performed by issuing a read or write on a **/dev/rlv***n* character special file for a logical volume. The read or write is processed by the file system SVC handler, which calls the LVDD **ddread** or **ddwrite** entry point. The **ddread** or **ddwrite** entry point transforms the character request into a block request. This is done by building a buffer for the request and calling the LVDD **ddstrategy** entry point.

Block I/O requests are performed by issuing a read or write on a block special file **/dev/lv***n* for a logical volume. These requests go through the SVC handler to the **bread** or **bwrite** block I/O kernel services. These services build buffers for the request and call the LVDD **ddstrategy** entry point. The LVDD **ddstrategy** entry point then translates the logical address to a physical address (handling mirroring and bad-block relocation) and calls the appropriate physical disk device driver.

On completion of the I/O, the physical disk device driver calls the **iodone** kernel service on the device interrupt level. This service then calls the LVDD I/O completion-handling routine. Once this is completed, the LVDD calls the **iodone** service again to notify the requester that the I/O is completed.

The LVDD is logically split into top and bottom halves. The top half contains the **ddopen**, **ddclose, ddread, ddwrite, ddioctl**, and **ddconfig** entry points. The bottom half contains the **ddstrategy** entry point, which contains block read and write code. This is done to isolate the code that must run fully pinned and has no access to user process context. The bottom half of the device driver runs on interrupt levels and is not permitted to page fault. The top half runs in the context of a process address space and can page fault.

Data Structures

The interface to the **ddstrategy** entry point is one or more logical **buf** structures in a list. The logical **buf** structure is defined in the **<sys/buf.h>** file and contains all needed information about an I/0 request, including a pointer to the data buffer. The **ddstrategy** entry point associates one or more (if mirrored) physical **buf** structures (or **pbufs**) with each logical **buf** structure and passes them to the appropriate physical device driver.

The physical **buf** structure (**pbuf**) is defined in the **<sys/dasd.h>** file. It is a standard **buf** structure with some additional fields. These fields are used by the LVDD to track the status of the physical requests that correspond to each logical I/O request. A pool of pinned **pbuf** structures is allocated and managed by the LVDD.

There is one device switch entry for each volume group defined on the system. Each volume group entry contains a pointer to the volume group data structure describing it.

Top Half of Logical Volume Device Driver

The top half of the LVDD contains the code that runs in the context of a process address space and can page fault. It contains the following entry points:

ddopen	Called by the file system when a logical volume is mounted, to open the logical volume specified.
ddclose	Called by the file system when a logical volume is unmounted, to close the logical volume specified.
ddconfig	Initializes data structures for the logical volume device driver.
ddread	Called by the **read** subroutine to translate character I/O requests to block I/O requests. This entry point verifies that the request is on a 512-byte boundary and is a multiple of 512 bytes in length. When a character request spans partitions or logical tracks (32 - 4K pages), the LVDD **ddread** routine breaks it into multiple requests. The routine then builds a buffer for each request, and passes it to the LVDD **ddstrategy** entry point, which handles logical block I/O requests. If the *ext* parameter is set (called by **readx** subroutine), the **ddread** entry point passes this parameter to the LVDD **ddstrategy** routine in the **b_options** field of the buffer header.
ddwrite	Called by the **write** subroutine to translate character I/O requests to block I/O requests. The LVDD **ddwrite** routine performs the same processing for a write request as the LVDD **ddread** routine does for read requests.
ddioctl	Supports the IOCINFO and XLATE operations, which return LVM configuration information.

Bottom Half of Logical Volume Device Driver

The bottom half of the device driver supports the **ddstrategy** entry point. This entry point processes all logical block requests and performs the following functions;

- Validates I/O requests.
- Checks requests for conflicts (such as overlapping block ranges) with requests currently in progress.
- Translates logical addresses to physical addresses.
- Handles mirroring and bad-block relocation.

The bottom half of the LVDD runs on interrupt levels and, as a result, is not permitted to page fault. The bottom half of the LVDD is divided into three layers as follows:

- Strategy
- Scheduler
- Physical.

Each logical I/O request passes down through the bottom three layers before reaching the physical disk device driver. Once the I/O is complete, the request returns back up through the layers to handle the I/O completion processing at each layer. Finally, control returns to the original requestor.

Strategy Layer

The strategy layer deals only with logical requests. The **ddstrategy** entry point is called with one or more logical **buf** structures. A list of **buf** structures for requests that are not blocked are passed to the second layer, the scheduler.

Scheduler Layer

The scheduler layer schedules physical requests for logical operations and handles mirroring and the mirror write consistency cache. For each logical request the scheduler layer schedules one or more physical requests. This involves translating logical addresses to physical addresses, handling mirroring, and calling the LVDD physical layer with a list of physical requests.

When a physical I/O operation is complete for one phase or mirror of a logical request, the scheduler initiates the next phase (if there is one). If no more I/O operations are required for the request, the scheduler calls the strategy termination routine. This routine notifies the originator that the request has been completed.

The scheduler also handles the mirror write consistency cache for the volume group. If a logical volume is using mirror write consistency (MWC), then requests for this logical volume are held within the scheduling layer until the MWC cache blocks can be updated on the target physical volumes.

Physical Layer

The physical layer of the LVDD handles startup and termination of the physical request. The physical layer calls a physical disk device driver's **ddstrategy** entry point with a list of **buf** structures linked together. In turn, the physical layer is called by the **iodone** kernel service when each physical request is completed.

This layer also performs bad-block relocation and detection/correction of bad blocks, when necessary. These details are thus hidden from the other two layers.

Interface to Physical Disk Device Drivers

Physical disk device drivers should adhere to the following criteria if they are to be accessed by the logical volume device driver:

- Disk block size must be 512 bytes.
- The physical disk device driver needs to accept a list of requests defined by **buf** structures which are linked together by the **av_forw** field in each **buf** structure.
- For unrecoverable media errors, physical disk device drivers need to set the following:
  - The **B_ERROR** flag on (defined in the **<sys/buf.h>** file) in the **b_flags** field.
  - The **b_error** field to E_MEDIA (defined in the **<sys/errno.h>** file).
  - The **b_resid** field to contain the number of bytes in the request that were not read or written successfully. The **b_resid** field is used to determine the block in error.

    Note: For write requests, the LVDD attempts to hardware-relocate the bad block. If this fails, then the block is software-relocated. For read requests, the information is recorded and the block is relocated on the next write request to that block.

- For a successful request that generated an excessive number of retries, the device driver can return good data. To indicate this situation it should set the following:
  - The **b_error** field should be set to ESOFT (defined in the **<sys/errno.h>** file).
  - The **b_flags** field should have the **B_ERROR** flag set on.
  - The **b_resid** field should be set to a count indicating the first block in the request that had excessive retries. This block is then relocated.
- The physical disk device driver needs to accept a request of one block with HWRELOC (defined in **<sys/lvdd.h>**) set on in the **b_options** field. This indicates that the device driver is to do a hardware relocation on this request. If the device driver does not support hardware relocation the following should be set:
  - The **b_error** field should be set to EIO (defined in the **<sys/errno.h>** file).
  - The **b_flags** field should have the **B_ERROR** flag set on.
  - The **b_resid** field should be set.
- The physical disk device driver should support the system dump interface as defined.
- The physical disk device driver must support write verification on an I/O request. Requests for write verification are made by setting the **b_options** field to WRITEV. This value is defined in the **<sys/lvdd.h>** file.

Logical Volumes and Bad Blocks

The physical layer of the LVDD initiates all bad-block processing and isolates all of the decision making from the physical disk device driver. This is done so that the physical disk device driver does not need to know anything about *mirroring*. Mirroring is the duplication of a physical partition that contains data.

Relocating Bad Blocks

The physical layer of the logical volume device driver (LVDD) checks each physical request to see if there are any known software-relocated bad blocks in the request. The LVDD determines if a request contains known software-relocated bad blocks by hashing the physical address. Then, a hash chain of the LVDD defects directory is searched to see if any bad-block entries are in the address range of the request.

If bad blocks exist in a physical request, the request is split into three separate pieces. The first piece contains any blocks up to the bad block. The second contains the relocated block (the relocated address is specified in the bad-block entry) of the defects directory. The third piece contains any blocks after the bad block to the end of the request. These separate pieces are processed sequentially.

Once the I/O for the first of the separated pieces has completed, the **iodone** kernel service calls the LVDD physical layer's termination routine (specified in the **b_done** field of the **buf** structure). The termination routine initiates I/O for the second piece of the original request (containing the relocated block), and then for the remaining (third) piece. Once the entire physical operation is completed, the appropriate scheduler's policy routine (in the second layer of the LVDD) is called to start the next phase of the logical operation.

Detecting and Correcting Bad Blocks

If a logical volume is mirrored, a newly detected bad block is fixed by relocating the block, reading the mirror, and writing the contents of the good mirror to the relocated block. With mirroring, the user need not even know when bad blocks are found. However, the physical disk device driver does in fact log permanent I/O errors so the user can determine the rate of media surface errors.

When a bad block is detected during I/O, the physical disk device driver sets the error fields in the **buf** structure to indicate that there was a media surface error. The physical layer of the LVDD then initiates any bad-block processing that must be done.

If the operation was a non-mirrored read, the block is not relocated because the data in the relocated block is not initialized until a write is performed to the block. To support this delayed relocation, an entry for the bad block is put into the LVDD defects directory and into the bad-block directory on disk. These entries contain no relocated block address and the status for the block is set to indicate that relocation is desired.

On each I/O request the physical layer checks whether there are any bad blocks in the request. If the request is a write and it contains a block that is in a `relocation-desired` state, the request is sent to the physical disk device driver with safe hardware relocation requested. If the request is a read, an I/O error is returned to the original requestor.

If the operation was for a mirrored read, a request to read one of the other mirrors is initiated. If the second read is successful, then the read is turned into a write request and the physical disk device driver is called with **safe** hardware relocation specified to fix the bad mirror.

If the hardware relocation fails or the device does not support safe hardware relocation, the physical layer of the LVDD attempts software relocation. At the end of each volume is a reserved area used by the LVDD as a pool of relocation blocks. When a bad block is detected and the disk device driver is unable to relocate the block, the LVDD picks the next unused block in the relocation pool and writes to this new location. A new entry is added to the LVDD defects directory in memory (and to the bad-block directory on disk) that maps the bad-block address to the new relocation block address. Any subsequent I/O requests to the bad-block address are routed to the relocation address.

Related Information

The **bread** kernel service, **bwrite** kernel service, **iodone** kernel service.

The **lvdd** special file.

The **buf** structure.

The **write** subroutine, **readx** subroutine.

Bad Block Relocation Policy, Understanding Volume Groups, The Vary-On and Vary-Off Process in *General Programming Concepts.*

Device Driver Classes, Device Driver Roles, Device Driver Structure, Logical Volume Storage Overview, Major and Minor Numbers, Understanding Block I/O Device Drivers, and Understanding Character I/O Device Driver in *Kernel Extensions and Device Support Programming Concepts.*

Chapter 8. Printer Subsystem

Understanding Embedded References in Printer Attribute Strings

Description

The attribute string retrieved by the **piocmdout**, **piogetstr**, and **piogetvals** subroutines can contain embedded references to other attribute strings or integers. The attribute string can also contain embedded logic that dynamically determines the content of the constructed string. This allows the constructed string to reflect the state of the formatter environment when one of these subroutines is called.

Embedded references and logic are defined with escape sequences that are placed at appropriate locations in the attribute string. The first character of each escape sequence is always the **%** (percent) character. This character indicates the beginning of an escape sequence. The second character (and sometimes subsequent characters) define the operation to be performed. The remainder of the characters (if any) in the escape sequence are operands to be used in performing the specified operation.

The escape sequences that can be specified in an attribute string are based on the **terminfo** parameterized string escape sequences for terminals. These escape sequences have been modified and extended for printers.

The attribute names that can be referenced by attribute strings are:

- The names of all attribute variables (which can be integer or string variables) defined to the **piogetvals** subroutine. When references are made to these variables, the **piogetvals**-defined versions are the values used.
- All other attributes names in the data base. These attributes are considered string constants.

Any attribute value (integer variable, string variable, or string constant) can be referenced by any attribute string. Consequently, it is important that the formatter ensure that the values for all the integer variables and string variables defined to the **piogetvals** subroutine are kept current.

The formatter must not assume that the particular attribute string whose name it specifies to the **piogetstr** or **piocmdout** subroutine does not reference certain variables. The attribute string is retrieved from the data base that is external to the formatter. The values in the data base represented by the string can be changed to reference additional variables without the formatter's knowledge.

Related Information

The **piocmdout** printer addition subroutine, **piogetstr** printer addition subroutine, **piogetvals** printer addition subroutine.

Printer Addition Management Subsystem: Programming Overview in *Kernel Extensions and Device Support Programming Concepts.*

initialize Subroutine

Purpose

Performs printer initialization.

Library

None (provided by the formatter).

Syntax

#include <piostruct.h>

int initialize ()

Description

The **initialize** subroutine is invoked by the formatter driver after the **setup** subroutine returns.

If the **–j** flag passed from the command line has a nonzero value (true), the **initialize** subroutine uses the **piocmdout** subroutine to send a command string to the printer. This action initializes the printer to the proper state for printing the file. Any variables referenced by the command string should be the attribute values from the database, overridden by values from the command line.

If the **–j** flag has a nonzero value (true), any necessary fonts should be downloaded.

Return Values

0	Indicates a successful operation.

If the **initialize** subroutine detects an error, it uses the **piomsgout** subroutine to invoke an error message. It then invokes the **pioexit** subroutine with a value of PIOEXITBAD. Note that if either the **piocmdout** or **piogetstr** subroutine detects an error, it automatically issues its own error messages and terminates the print job.

Related Information

The **piocmdout** subroutine.

Subroutines for Writing a Printer Formatter in *Kernel Extensions and Device Support Programming Concepts.*

lineout Subroutine

Purpose

Formats a print line.

Library

None (provided by the print formatter).

Syntax

#include <piostruct.h>

int lineout (*fileptr***)**
**FILE ****fileptr*;

Parameter

fileptr	Specifies a file structure for the input data stream.

Description

The **lineout** subroutine is invoked by the formatter driver only if the **setup** subroutine returns a non–NULL pointer. This subroutine is invoked for each line of the document being formatted. The **lineout** subroutine reads the input data stream from the *fileptr* parameter. It then formats and outputs the print line until it recognizes a situation that causes vertical movement on the page.

The **lineout** subroutine should process all characters to be printed and all printer commands related to horizontal movement on the page.

The **lineout** subroutine should not output any printer commands that cause vertical movement on the page. Instead, it should update the **vpos** (new vertical position) variable pointed to by the **shars_vars** structure that it shares with the formatter driver to indicate the new vertical position on the page. It should also refresh the **shar_vars** variables for vertical increment and vertical decrement (reverse line feed) commands.

When the **lineout** subroutine returns, the formatter driver sends the necessary commands to the printer to advance to the new vertical position on the page. This position is specified by the **vpos** variable. The formatter driver automatically handles top and bottom margins, new pages, initial pages to be skipped, and progress reports to the **qdaemon** daemon.

The following conditions can cause vertical movements:

- Linefeed control character or variable line feed control sequence
- Vertical-tab control character
- Formfeed control character
- Reverse linefeed control character
- A line too long for the printer that wraps to the next line.

Other conditions unique to a specific printer also cause vertical movement.

Return Values

Upon successful completion, the **lineout** subroutine returns the number of bytes processed from the input data stream. It excludes the end-of-file character and any control characters or escape sequences that result only in vertical movement on the page (for example, line feed or vertical tab).

If a value of 0 is returned and the value in the **vpos** variable pointed to by the **shars_vars** structure has not changed, or there are no more data bytes in the input data stream, the formatter driver assumes that printing is complete.

If the **lineout** subroutine detects an error, it uses the **piomsgout** subroutine to issue an error message. It then invokes the **pioexit** subroutine with a value of PIOEXITBAD. Note that if either the **piocmdout** or **piogetstr** subroutine detects an error, it automatically issues its own error messages and terminates the print job.

Related Information

The **setup** subroutine.

Subroutines for Writing a Printer Formatter in *Kernel Extensions and Device Support Programming Concepts.*

passthru Subroutine

Purpose

Passes through the input data stream without modification or formats the input data stream without assistance from the formatter driver.

Library

None (provided by the print formatter).

Syntax

#include <piostruct.h>

int passthru ()

Description

The **passthru** subroutine is invoked by the formatter driver only if the **setup** subroutine returned a NULL pointer. If this is the case, the **passthru** subroutine is invoked (instead of the **lineout** subroutine) for one of these two reasons:

- The input data stream is to be passed through without modification.
- Formatting is done without the help of the formatter driver to handle vertical spacing.

Even if the data is being passed through from input to output without modification, a formatter program is used to initialize the printer before printing the file and to restore it to a known state afterward. However, gathering accounting information for an unknown data stream being passed through is difficult, if not impossible.

The **passthru** subroutine can also be used to format the input data stream if no help from the formatter driver for vertical spacing is needed. For example, if the only formatting to be done is to add a carrier-return control character to each linefeed control character, the **passthru** subroutine provides this simple task. The **passthru** subroutine can also count line feeds and form feeds to keep track of the page count. These counts can then be reported to the **log_pages** status subroutine, which is provided by the spooler.

Return Values

0	Indicates a successful operation.

If the **passthru** subroutine detects an error, it uses the **piomsgout** subroutine to issue an error message. It then invokes the **pioexit** subroutine with a value of PIOEXITBAD. Note that if the **passthru** subroutine calls the **piocmdout** subroutine or the **piogetstr** subroutine and either of these detects an error, then the subroutine that detects the error automatically issues its own error message and terminates the print job.

Related Information

The **lineout** subroutine, **setup** subroutine.

Subroutines for Writing a Printer Formatter in *Kernel Extensions and Device Support Programming Concepts.*

piocmdout Subroutine

Purpose

Outputs an attribute string for a printer formatter.

Library

None (linked with the **pioformat** formatter driver).

Syntax

#include <piostruct.h>

piocmdout (*attrname*, *fileptr*, *passthru*, **NULL)**
char **attrname*;
FILE **fileptr*;
int *passthru*;

Parameters

attrname	Points to a two–character attribute name for a string. The attribute name must be defined in the database and can optionally have been defined to the **piogetvals** subroutine as a variable string. The attribute should not be one that has been defined to the **piogetvals** subroutine as an integer.
fileptr	Specifies a file pointer for the input data stream. If the **piocmdout** routine is called from the **lineout** formatter routine, the *fileptr* value should be the *fileptr* passed to the **lineout** routine as a parameter. Otherwise, the *fileptr* value should be **stdin**. If the *passthru* parameter is 0 (zero), the *fileptr* parameter is ignored.
passthru	Specifies the number of bytes to be passed to standard output unmodified from the input data stream specified by the *fileptr* parameter. This occurs when the **%x** escape sequence is found in the attribute string or in a string included by the attribute string. If no **%x** escape sequence is found, the specified number of bytes is read from the input data stream and discarded. If no bytes are to be passed through, the *passthru* parameter should be 0.

Description

The **piocmdout** subroutine retrieves the specified attribute string from the Printer Attribute database and outputs the string to standard output. In the course of retrieval, this subroutine also resolves any logic and any embedded references to other attribute strings or integers.

The *fileptr* and *passthru* parameters are used to pass data that the formatter does not need to scan (for example, graphics data) from the input data stream to standard output.

Return Values

Upon successful completion, the **piocmdout** subroutine returns the length of the constructed string.

If the **piocmdout** subroutine detects an error, it issues an error message and terminates the print job.

Related Information

The **piogetvals** subroutine.

Embedded References in Printer Attribute Strings, Subroutines for Printer Formatters in *Kernel Extensions and Device Support Programming Concepts.*

pioexit Subroutine

Purpose

Exits from a printer formatter.

Library

None (linked with the **pioformat** formatter driver).

Syntax

#include <piostruct.h>

void pioexit (*exitcode*)
int *exitcode*;

Parameter

exitcode	Specifies whether the formatting operation completed successfully. A value of PIOEXITGOOD indicates that the formatting completed normally. A value of PIOEXITBAD indicates that an error was detected.

Description

The **pioexit** subroutine should be used by printer formatters to exit when either formatting is complete or an error has been detected. This subroutine is supplied by the formatter driver.

The **pioexit** subroutine has no return values.

Related Information

Subroutines for Printer Formatters in *Kernel Extensions and Device Support Programming Concepts*.

piogetopt Subroutine

Purpose

Overlays default flag values from the database colon file with override values from the command line.

Library

None (linked with the **pioformat** formatter driver).

Syntax

#include <piostruct.h>

int piogetopt (*argc*, *argv*, *optstring*, **NULL)**
int *argc*;
char **argv* **[ ],** **optstring*;

Parameters

argc	Same as the *argc* parameter received by the formatter's **setup** routine when it was called by the formatter driver.
argv	Same as the *argv* parameter received by the formatter's **setup** routine when it was called by the formatter driver.
optstring	Specifies a string of flag letters for flags that the formatter expects from the command line. By convention, each flag specified on the command line and passed to a formatter by the formatter driver must have an argument. This means that each flag letter in the *optstring* string must be followed by a : (colon) to indicate that the flag has an argument.

Description

The **piogetopt** subroutine should be used by a printer formatter's **setup** routine to perform these three tasks:

- Parse the command line flags.
- Convert the flag arguments, as needed, to the data types specified in the array of **attrparms** structures previously passed to the **piogetvals** subroutine.
- Overlay the default flag arguments with values from the data base.

The **piogetopt** subroutine is supplied by the formatter driver.

The database attribute names for flags with integer arguments must have previously been defined to the formatter driver with the **piogetvals** subroutine. Based on the information that was provided to the **piogetvals** subroutine, the **piogetopt** subroutine takes these three actions:

- Recognizes each flag argument that needs to be converted to an integer value.
- Converts the argument string to an integer value using the conversion method specified to the **piogetvals** subroutine.
- Regardless of the data type (integer variable, string variable, or string constant) overlays the default value from the database.

Return Values

0	Indicates successful completion.

If the **piogetopt** subroutine detects an error, it issues an error message and terminates the print job.

Related Information

The **piogetvals** subroutine, **setup** subroutine.

Subroutines for Printer Formatters in *Kernel Extensions and Device Support Programming Concepts.*

piogetstr Subroutine

Purpose

Retrieves an attribute string for a printer formatter.

Library

None (linked with the **pioformat** formatter driver).

Syntax

#include <piostruct.h>

piogetstr (*attrname*, *bfrptr*, *bufsiz*, **NULL)**
char **attrname*, **bufptr*;
int *bufsiz*;

Parameters

attrname	Points to a two–character attribute name for a string. The attribute name must be defined in the database. It may optionally have been defined to the **piogetvals** subroutine as a variable string. The attribute should not be one that has been defined to the **piogetvals** subroutine as an integer.
bufptr	Points to where the constructed attribute string is to be stored.
bufsize	Specifies the amount of memory that is available for storage of the string.

Description

The **piogetstr** subroutine retrieves the specified attribute string from the Printer Attribute database and returns the string to the caller. In the course of retrieval, this subroutine also resolves any logic and any embedded references to other attribute strings or integers.

Return Values

Upon successful completion, the **piogetstr** subroutine returns the length of the constructed string. The null character placed at the end of a constructed string by the **piogetstr** subroutine is not included in the length.

If the **piogetstr** subroutine detects an error, it issues an error message and terminates the print job.

Related Information

The **piogetvals** subroutine.

Embedded References in Printer Attribute Strings, Subroutines for Printer Formatters in *Kernel Extensions and Device Support Programming Concepts.*

piogetvals Subroutine

Purpose

Initializes a copy of Printer Attribute database variables for a printer formatter.

Library

None (linked with the **pioformat** formatter driver).

Syntax

#include <piostruct.h>

int piogetvals (*attrtable***, NULL)**
struct attrparms *attrtable* **[ ];**

Parameter

attrtable	Points to a table of variables and their characteristics. The table is an array of **attrparms** structures, as defined in the **piostruct.h** header file.

The second parameter is reserved for future use. It should be a NULL pointer.

Description

The **piogetvals** subroutine provides a way for a printer formatter's **setup** routine to define a list of printer attribute variables (and their characteristics) to the formatter driver. This routine, which is supplied by the formatter driver, allocates storage for the requested variables and uses the Printer Attribute database colon file to arrive at initial values.

The variables defined by the **piogetvals** subroutine are copies of variables in the database, and are used to hold current values of those (database) variables. After the **piogetvals** subroutine returns pointers to each of the variables, the characteristics and memory location of each variable is known to both the formatter and the formatter driver. Subsequent changes to printer attribute values (made by the formatter while formatting an input data stream) are made to the newly defined variables, not to the database values. As a result of this scheme, the formatter driver always has access to the current value of each variable, but does not itself ever modify them.

The caller requests variables by filling in entries (an attribute name, its data type, and other characteristics) in the table pointed to by the *attrtable* parameter. For each entry, the **piogetvals** subroutine retrieves the requested attribute string in the Printer Attribute data base and converts it, if necessary, into an actual value. The **piogetvals** subroutine then allocates memory for each of the variables, places the initial values there, and stores information about the variable (its name, data type, and memory location) in storage accessible to the **piogetopt**, **piocmdout**, and **piogetstr** subroutines.

Printer Attribute Variables

A Printer Attribute database is a colon file containing printer attribute values, which can be overridden at the time a print job is requested. These attributes can be constants or may be expressions with unresolved references to other attributes in them. These references are resolved before a database attribute is used to fill in the value of a requested variable.

Database attribute values, which are stored in the database as ASCII strings, have possible data types of string constant (the default), integer variable, or string variable. The requested variables should be either integers or strings. String variables are used primarily for strings that the formatter may need to modify during its processing. NULL characters have no special significance and are permissible within variable strings.

Data types for the requested variables are specified in the array of the **attrparms** structures pointed to by the *attrtable* parameter and are not specified at all in the Printer Attribute database. This means that for database values used exclusively by the formatter, only the formatter knows the actual data type of each value. Thus the formatter uses the **piogetvals** routine in part to inform the formatter driver of the actual data type for database values that are not the default data type.

Converting a Database Attribute String to an Actual Value

Converting a database attribute string to an actual value involves two aspects. First the **piogetvals** routine resolves any logic and any embedded references to other attribute strings, which yields a resolved string variable. Secondly, the data type of the requested variable must be checked. If this data type specifies a character string, then the resolved string is the final value, and it is stored in the memory allocated for it.

However, if the specified data type is integer variable, then the resolved string is converted to an integer. In this case, the *attrtable* entry for the attribute string is checked to determine how this conversion is to be performed. Either use the **atoi** subroutine for this purpose, or provide a pointer to a lookup table. After being converted to an integer, the value is stored in the memory allocated for it.

Using the **piogetvals** subroutine to convert database strings to integers as specified by the *attrtable* entries provides a table–driven procedure for the conversions. It also informs the formatter driver which values are integers and how strings that represent the integers can be converted into integer values. The **piogetopt**, **piocmdout**, and **piogetstr** subroutines assume that the formatter has used the **piogetvals** subroutine to provide this information about the variables to the formatter driver.

When a formatter subsequently calls either the **piocmdout** subroutine or the **piogetstr** subroutine to access a string from the database, a global list of variables defined by the **piogetvals** subroutine is checked by the subroutine to see if the desired string has been defined. If so, then the value of the variable is taken from the memory location specified in the global list. If not, then the Printer Attribute database is consulted for the correct attribute string. Either the **piocmdout** or **piogetstr** subroutine scans the string to resolve any logic and any references to other strings or integers. Thus the characteristics and memory locations of the variables, as remembered by the **piogetvals** subroutine, are used to obtain the current values of the variables.

Return Values

0	Indicates a successful operation.

If the **piogetvals** subroutine detects an error, it issues an error message and terminates the print job.

Related Information

The **piocmdout** printer addition subroutine, **piogetopt** printer addition subroutine, **piogetstr** printer addition subroutine, **setup** printer addition subroutine.

Embedded References in Printer Attribute Strings, Subroutines for Printer Formatters in *Kernel Extensions and Device Support Programming Concepts.*

piomsgout Subroutine

Purpose

Sends a message from a printer formatter.

Library

None (linked with the **pioformat** formatter driver).

Syntax

void piomsgout (*msgstr***)**
char ****msgstr*;**

Parameter

msgstr	Points to the string of message text to be sent.

Description

The **piomsgout** subroutine should be used by printer formatters to send a message to the print job submitter, usually when an error is detected. This subroutine is supplied by the formatter driver.

If the formatter is running under the spooler, the message is displayed on the submitter's terminal if the submitter is logged on. Otherwise, the message is mailed to the submitter. If the formatter is not running under the spooler, the message is sent as standard error output.

The **piomsgout** subroutine has no return values.

Related Information

Subroutines for Printer Formatters in *Kernel Extensions and Device Support Programming Concepts.*

restore Subroutine

Purpose

Restores the printer to its default state.

Library

None (provided by the print formatter)

Syntax

#include <piostruct.h>

int restore ()

Description

The **restore** subroutine is invoked by the formatter driver after either the **lineout** subroutine or the **passthru** subroutine has reported that printing has completed.

If the **–J** flag passed from the command line has a nonzero value (true), the **initialize** subroutine should use the **piocmdout** subroutine to send a command string to the printer to restore the printer to its default state. This default state is defined by the attribute values in the database. Any variables referenced by the command string should be values from the database that have not been overridden by values from the command line. This can be accomplished by placing a **%o** escape sequence at the beginning of the command string.

Return Values

0	Indicates a successful operation.

If the **restore** subroutine detects an error, it uses the **piomsgout** subroutine to issue an error message. The **restore** subroutine then invokes the **pioexit** subroutine with a value of PIOEXITBAD. If either the **piocmdout** or **piogetstr** subroutine detects an error, then the subroutine that detects the error issues an error message and terminates the print job.

Related Information

The **lineout** subroutine, **passthru** subroutine.

Subroutines for Writing a Printer Formatter in *Kernel Extensions and Device Support Programming Concepts.*

setup Subroutine

Purpose

Performs setup processing for the print formatter.

Library

None (provided by the print formatter).

Syntax

#include <piostruct.h>

struct shar_vars *setup (*argc*, *argv*, *passthru***)**
unsigned *argc*;
char **argv*[ ];
int *passthru*;

Description

The **setup** subroutine performs these three tasks:

- Invokes the **piogetvals** subroutine to initialize the database variables that the formatter uses.
- Processes the command line flags using the **piogetopt** subroutine.
- Validates the input parameters from the database and the command line.

The **setup** subroutine should not send commands or data to the printer since the formatter driver performs additional error checking when the **setup** subroutine returns.

Parameters

argc	Specifies the number of formatting arguments from the command line (including the command name).
argv	Points to a list of pointers to the formatting arguments.
passthru	Indicates whether the input data stream should be formatted (the *passthru* parameter is 0), or passed through without modification (the value of the *passthru* parameter equals1). The value for this parameter is the argument value for the **–#** flag specified to the **pioformat** formatter driver. If the **–#** flag is not specified, the *passthru* parameter value is 0.

Return Values

Upon successful completion, the **setup** subroutine returns one of these two pointers:

- A pointer to a **shar_vars** structure that contains pointers to initialized vertical spacing variables. These variables are shared with the formatter driver, which provides vertical page movement.
- A NULL pointer, which indicates that the formatter handles its own vertical page movement or that the input data stream is to be passed through without modification. Vertical page movement includes top and bottom margins, new pages, initial pages to be skipped, and progress reports to the **qdaemon** daemon.

Returning a pointer to a **shar_vars** structure causes the formatter driver to invoke the formatter's lineout function for each line to be printed. Returning a NULL pointer causes the formatter driver to invoke the formatter's *passthru* function once instead.

If the **setup** subroutine detects an error, it uses the **piomsgout** subroutine to issue an error message. The **setup** subroutine then invokes the **pioexit** subroutine with a value of PIOEXITBAD. Note that if the **piogetvals**, **piogetopt**, **piocmdout**, or **piogetstr** subroutine detects an error, it automatically issues its own error message and terminates the print job.

Related Information

The **piogetvals** subroutine.

Subroutines for Writing a Printer Formatter in *Kernel Extensions and Device Support Programming Concepts.*

Chapter 9. SCSI Subsystem

CD-ROM SCSI Device Driver

Purpose

Supports the CD-ROM (compact-disk read-only memory) device driver.

Syntax

#include <sys/devinfo.h>
#include <sys/cdrom.h>
#include <sys/scsi.h>

Description

The **/dev/cd***n* and **/dev/rcd***n* special files provide block and character (raw) access to disks in the CD-ROM drives. Compact disks are read-only media that can store large amounts of data. Block access to compact disks is through the special files **/dev/cd0,** .... Character access is provided through the special files **/dev/rcd0,** ... .

The **r** prefix on a special file name means that the drive is accessed as a raw device rather than a block device. Performing raw I/O with a compact disk requires that all data transfers be in multiples of the compact disk logical block length. Also, all **lseek** subroutines made to the raw CD-ROM device driver must result in a file pointer value that is a multiple of the specified logical block size.

Note: For the compact disk type supported, the logical block length is 512 bytes. Configuration information for the CD-ROM device is defined in the **<sys/cdrom.h>** file in the **cdrom_dds_df** structure.

Device-Dependent Subroutines

Most CD-ROM operations are implemented using the **open**, **read**, and **close** subroutines. However, for some purposes, use of the **openx** subroutine is required.

The open and close Subroutines

The **openx** subroutine is intended primarily for use by the diagnostic commands and utilities. Appropriate authority is required for execution. Attempting to execute this subroutine without the proper authority returns a value of -1 and the **errno** global variable is set to EPERM.

The **openx** subroutine allows the device driver to enter Diagnostic mode and disables command retry logic. This allows for execution of **ioctl** operations that perform special functions associated with diagnostic processing. Other **openx** capabilities (such as forced opens and retained reservations) are also available.

The *ext* parameter passed to the **openx** subroutine selects the operation to be used for the target device. The acceptable values for the *ext* parameter are defined the **<sys/scsi.h>** file. This parameter accepts any combination of the following three flag values logically ORd together:

- **SC_DIAGNOSTIC**

  Places the selected device in Diagnostic mode. This mode is singularly entrant. When a device is in Diagnostic mode, SCSI operations are performed during **open** or **close** operations, and error logging is disabled. In Diagnostic mode, only the **close** and **ioctl** operations are accepted. All other device-supported subroutines return a value of -1, with the **errno** global variable set to EACCES.

  A device can be opened in Diagnostic mode only if the target device is not currently opened. If an attempt is made to open a device in Diagnostic mode and the target device is already open, a value of -1 is returned and the **errno** global variable is set to EACCES.

- **SC_FORCED_OPEN**

  Forces a Bus Device Reset regardless of whether another initiator has the device reserved. The SCSI Bus Device Reset is sent to the device before the **open** sequence begins. Otherwise, the open executes normally.

- **SC_RETAIN_RESERVATION**

  Retains the reservation of the device after a **close** operation by not issuing the release. This flag prevents other initiators from using the device unless they break the host machine's reservation.

  Unlike other SCSI device drivers, the CD-ROM driver assumes that users do not need exclusive access to the device since its access mode is read-only. As a result, the **reserve_lock** field is initialized to FALSE for the default database. This field is contained in the **cdrom_dds_df** structure. The **cdrom_dds_df** structure is defined in the **<sys/cdrom.h>** file. No initiator can lock others out from accessing the CD-ROM.

  Note: The SC_RETAIN_RESERVATION flag does not ordinarily have effect. However, the user can alter the configuration database to make this flag effective using the **smit** command.

SCSI Options to the **openx** subroutine gives more specific information on these operations.

The ioctl Subroutine

The following operations are available for use with the CD-ROM device driver.

IOCINFO Returns a **devinfo** structure as defined in the **<sys/devinfo.h>** file.

CDIOCMD Allows SCSI commands to be issued directly to the attached CD-ROM device. For this operation, the device must be opened in Diagnostic mode. The CDIOCMD operation parameter is the address of a **sc_iocmd** structure. This structure is defined in the **<sys/scsi.h>** file.

If this command is attempted on a device not in Diagnostic mode, a value of -1 is returned and the **errno** global variable is set to EACCES. Refer to the Small Computer System Interface (SCSI) specification for the applicable device for issuing the proper parameters.

CDIORDSE Provides the means for issuing a **read** command and obtaining the target device sense data on an error. Diagnostic mode is not required when using this command. If this operation returns a –1 and the **status_validity** field has the SC_VALID_SENSE flag set, then valid sense data has been returned. Otherwise, target-sense data is omitted. Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

The CDIORDSE operation parameter is the address of a **sc_rdwrt** structure. This structure is defined in the **<sys/scsi.h>** file.

Error Conditions

In addition to those errors listed, **ioctl**, **open**, and **read** subroutines against this device fail in the following circumstances:

EACCES	Indicates that an attempt was made to open a device already in Diagnostic mode.
EACCES	Indicates that a subroutine other than **ioctl** or **close** was attempted while in Diagnostic mode.
EBUSY	Indicates that the target device is reserved by another initiator.
EINVAL	Indicates that the device has been opened with a mode other than read-only mode.
EINVAL	Indicates that the read passed an *nbyte* parameter that is not a multiple of the block size.
EINVAL	Indicates that a sense data buffer length greater than 255 is invalid for a CDIOCMD operation.
EINVAL	Indicates that a data buffer length greater than that specified in the **sc_maxrequest** field is invalid for a CDIOCMD operation.
EMFILE	Indicates an attempt to open a SCSI adapter that already has the maximum permissible number of opened devices
ENOTREADY	Indicates that there is no compact disk in the drive.
EPERM	Indicates that the subroutine attempted requires appropriate authority.
ESTALE	Indicates that the CD-ROM disk has been ejected (without first being closed by the user) and then either re-inserted or replaced with a second disk.
ETIMEDOUT	Indicates that a command has exceeded the given timer value.

Reliability and Serviceability Information

Errors returned from CD-ROM devices are categorized by the list described below:

GOOD COMPLETION	Indicates that the command completed successfully.
RECOVERED ERROR	Indicates that the command was successful after some recovery applied.
NOT READY	Indicates that the logical unit is offline.
MEDIUM ERROR	Indicates that the command terminated with a non-recoverable error condition.
HARDWARE ERROR	Indicates that the non-recoverable hardware failure during command execution or during a self test.
ILLEGAL REQUEST	Indicates illegal command or command parameter.
UNIT ATTENTION	Indicates that the CD-ROM changed or device has been reset or powered on.
ABORTED COMMAND	Indicates that the CD-ROM device aborted the command.
ADAPTER ERRORS	Indicates that the SCSI adapter indicated a CD-ROM drive error.

Medium, hardware, aborted command, and adapter errors from the above are logged every time they occur. If recovery at the device driver level is successful, the error is logged as temporary. Otherwise, it is logged as permanent. If recovery at the hardware level is successful, the error is not be reported to the driver and is therefore logged.

Regardless of which error is encountered, the following fields in the error record have the same value:

Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Inst_Causes	None.
Inst_Actions	None.
Detail_Data	Equal to 168, 11 to indicate HEX.

The **Detail_Data** field contains the **sc_error_log_df** structure defined in the **<sys/scsi.h>** file.

Error Record Fields for Permanent CD-ROM Medium Errors

Field values in the error record specific to permanent CD-ROM medium errors are:

Comment	Equal to CD-ROM permanent media error.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to E801, indicating a optical disk operation error.
Prob_Causes	Equal to 5004, indicating the optical disk.
User_Causes	Equal to 5100, indicating the media is defective.
User_Actions	Equal to 1601, indicating the removable media should be replaced and retried.
Fail_Causes	Equal to E800 or 6312, which are both optical disk drive errors.
Fail_Actions	Equal to 0000 to perform problem determination procedures.

The 128-byte request-sense data field is filled in with the complete request-sense data and padded with zeros to fill out the field. Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Error Record Fields for Recoverable CD-ROM Medium Errors

Field values in the error record specific to recoverable CD-ROM medium errors are:

Comment	Equal to CD-ROM temporary media error.
Err_Type	Equal to TEMP, indicating a temporary failure.
Err_Desc	Equal to E801, indicating a optical disk operation error.
Prob_Causes	Equal to 5004, indicating the optical disk.
User_Causes	Equal to 5100, indicating the media is defective.
User_Actions	Equal to 1601 or 0000, indicating the removable media should be replaced and retried and problem determination procedures should be performed, respectively.
Fail_Causes	Equal to E800 or 6312, which are both optical disk drive errors.
Fail_Actions	Equal to 1601 and 0000, indicating the removable media should be replaced and retried, and problem determination procedures should be performed, respectively.

The 128-byte request-sense data field is filled in with the complete request-sense data and padded with zeros to fill out the field. Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Error Record Fields for Permanent CD-ROM Hardware and Hard-Aborted Command Errors

Field values in the error record specific to permanent CD-ROM hardware errors and hard-aborted command errors are:

Comment	Equal to a CD-ROM permanent error.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to E801, indicating a optical disk drive error.
Prob_Causes	Equal to 6312, indicating the optical disk drive.
User_Causes	None.
User_Actions	None.
Fail_Causes	Equal to 6312 indicating the optical disk drive.
Fail_Actions	Equal to 0000 to perform problem determination procedures.

The 128-byte request-sense data field is filled in with the complete request-sense data and padded with zeros to fill out the field. Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Error Record Fields for Recoverable CD-ROM Hardware and Hard-Aborted Command Errors

Field values in the error record specific to recoverable CD-ROM hardware errors and hard-aborted command errors are:

Comment	Equal to a CD-ROM temporary error.
Err_Type	Equal to TEMP, indicating a temporary failure.
Err_Desc	Equal to E801, indicating a optical disk operation error.
Prob_Causes	Equal to 6312, indicating the optical disk drive.
User_Causes	None.
User_Actions	None.
Fail_Causes	Equal to 6312 indicating the optical disk drive.
Fail_Actions	Equal to 0000 to perform problem determination procedures.

The 128-byte request-sense data field will be filled in with the complete request-sense data and padded with zeros to fill out the field. Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Error Record Fields for Permanent Errors Returned from the SCSI Adapter

Field values in the error record specific to permanent errors returned from the SCSI adapter are:

Comment	Equal to an adapter-detected permanent CD-ROM error.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to E801, indicating a optical disk operation error.
Prob_Causes	Equal to 6312 and 3451, indicating the optical disk drive or the device cable, respectively.
User_Causes	None.
User_Actions	None.
Fail_Causes	Equal to 6312 indicating the optical disk drive.
Fail_Actions	Equal to 0000 to perform problem determination procedures.

The 128-byte request-sense data field is filled in with zeros for this error since there is no request-sense data available.

Error Record Fields for Recovered Errors Returned from the SCSI Adapter

Field values in the error record specific to recovered errors returned from the SCSI adapter are:

Comment	Equal to an adapter-detected temporary CD-ROM error.
Err_Type	Equal to TEMP, indicating a temporary failure.
Err_Desc	Equal to E801, indicating a optical disk operation error.
Prob_Causes	Equal to 6312 and 3451, indicating the optical disk drive or the device cable, respectively.
User_Causes	None.
User_Actions	None.
Fail_Causes	Equal to 6312 indicating the optical disk drive.
Fail_Actions	Equal to 0000 to perform problem determination procedures.

The 128-byte request-sense data field will be filled in with zeros for this error since there is no request-sense data available.

Error Record Fields for Errors Recovered at the Device Level

Field values in the error record specific to errors recovered at the device level are:

Comment	Equal to a CD-ROM device-recovered error.
Err_Type	Equal to TEMP, indicating a temporary failure.
Err_Desc	Equal to 1611, indicating an impending storage subsystem failure.
Prob_Causes	Equal to 6312 or 5004, indicating the optical disk drive or the optical disk drive.
User_Causes	Equal to 5100, indicating the media is defective.
User_Actions	Equal to 1601 and 0000, indicating the removable media should be replaced and the operation retried and problem determination should be performed, respectively.
Fail_Causes	Equal to E800 and 6312, indicating the optical disk or the optical disk drive, respectively.
Fail_Actions	Equal to 1601 and 0000, indicating the removable media should be replaced and the operation retried or problem determination procedures should be performed, respectively.

The 128 byte request-sense data field is filled in with the complete request-sense data and padded with zeros to fill out the field.

Error Record Fields for Unknown Errors

Field values in the error record specific to unknown errors, and conditions that should never occur are:

Comment	Equal to CD-ROM unknown error.
Err_Type	Equal to UNKN error.
Err_Desc	Equal to FE00, indicating the error is undetermined.
Prob_Causes	Equal to 5004, 6312., and 3300, indicating the optical disk, the optical disk drive, and the adapter, respectively.
User_Causes	None.
User_Actions	None.
Fail_Causes	Equal to FFFF unknown.
Fail_Actions	Equal to 2000, indicating review detail data.

The 128-byte request-sense data field is filled in with zeros for this error since there is no request-sense data available. Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Files

/dev/cd0, **/dev/cd1**, ...

/dev/rcd0, **/dev/rcd1**, ...

Related Information

The **smit** command.

The **cdrom.h** special file.

The **open** subroutine, **close** subroutine, **read** subroutine, **write** subroutine, **ioctl** subroutine.

The SCSI Adapter Device Driver, The SCSI Device Driver-SCSI Adapter Device Driver Interface.

Special File Overview in *Files Reference.*

Device Driver Concepts Overview, SCSI Subsystem Programming Introduction in *Kernel Extensions and Device Support Programming Concepts.*

Understanding Block I/O Device Drivers, Understanding I/O Access through Special Files, Understanding Major and Minor Numbers, Understanding Pseudo-Devices, Understanding the Device Switch Table in *Kernel Extensions and Device Support Programming Concepts*

rmt SCSI Device Driver

Purpose

Supports the sequential access bulk storage medium device driver.

Syntax

#include <sys/devinfo.h>
#include <sys/scsi.h>
#include <sys/tapedd.h>

Description

The /**dev/rmt0**, ..., /**dev/rmt255** special files provide access to magnetic tapes. Magnetic tapes are used primarily for backup, file archives, and other offline storage.

Note: Configuration information for the device is contained in the **tape_device_df** structure, as defined in the **<tapedd.h>** header file.

Device-Dependent Subroutines

Most tape operations are implemented using the **open**, **read**, **write**, and **close** subroutines. However, the **openx** subroutine must be used if the device is to be opened in Diagnostic mode.

The open and close Subroutines

The **openx** subroutine is intended for use by the diagnostic commands and utilities. Appropriate authority is required for execution. Attempting to execute this subroutine without the proper authority returns a value of –1 and the **errno** global variable is set to EPERM.

The **openx** subroutine allows the device driver to enter Diagnostic mode and disables command retry logic. This allows for execution of **ioctl** operations that perform special functions associated with diagnostic processing. Other **openx** capabilities (such as forced opens and retained reservations) are also available.

The *ext* parameter passed to the **openx** subroutine selects the operation to be used for the target device. The *ext* parameter is defined in the **<sys/scsi.h>** file. This parameter can contain any combination of the following flag values logically ORd together:

- **SC_DIAGNOSTIC**

  Places the selected device in Diagnostic mode. This mode is singularly entrant. When a device is in Diagnostic mode, SCSI operations are performed during **open** or **close** operations, and error logging is disabled. In Diagnostic mode, only the **close** and **ioctl** operations are accepted. All other device-supported subroutines return a –1, with the **errno** global variable set to EACCES.

  A device can be opened in Diagnostic mode only if the target device is not currently opened. If an attempt is made to open a device in Diagnostic mode and the target device is already open, a value of –1 is returned and the **errno** global variable is set to EACCES.

- **SC_FORCED_OPEN**

  Forces a bus device reset (BDR) regardless of whether another initiator has the device reserved. The SCSI bus device reset is sent to the device before the **open** sequence begins, otherwise, the open executes normally.

- **SC_RETAIN_RESERVATION**

  Retains the reservation of the device after a **close** operation by not issuing the release. This flag prevents other initiators from using the device unless they break the host machine's reservation.

SCSI options to the **openx** subroutine gives more specific information on the open operations.

The ioctl Subroutine

The STIOCMD **ioctl** operation provides the means for sending SCSI commands directly to a tape device. This allows an application to issue specific SCSI commands that are not directly supported by the tape device driver.

To use the STIOCMD operation, the device must be opened in Diagnostic mode. If this command is attempted while the device is not in Diagnostic mode, a value of –1 is returned and the **errno** global variable is set to EACCES. The STIOCMD operation passes the address of a **sc_iocmd** structure. This structure is defined in the **<sys/scsi.h>** file.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for issuing the proper parameters.

Error Conditions

In addition to those errors listed, **ioctl**, **open**, **read**, and **write** subroutines against this device fail in the following circumstances:

EAGAIN	Indicates that an attempt was made to open a device that was already open.
EACCES	Indicates that a diagnostic command was issued to a device not in Diagnostic mode.
EBUSY	Indicates that the target device is reserved by another initiator.
EINVAL	Indicates that a value of O_APPEND is supplied as the mode in which to open.
EINVAL	Indicates that the *nbyte* parameter supplied by a **read** or **write** operation is not a multiple of the block size.
EINVAL	Indicates that a parameter to an **ioctl** operation is invalid.
EINVAL	Indicates that the requested **ioctl** operation is not supported on the current device.
EIO	Indicates that the device could not space forward or reverse the number of records specified by the **st_count** field before encountering an EOM (end of media) or a file mark.
EMEDIA	Indicates that the tape device has encountered an unrecoverable media error.

EMFILE	Indicates that an **open** operation was attempted for a SCSI adapter that already has the maximum permissible number of open devices.
ENXIO	Indicates that there was an attempt to write to a tape that is at EOM.
ENOTREADY	Indicates that there is no tape in the drive or the drive is not ready.
EPERM	Indicates that this subroutine requires appropriate authority.
ETIMEDOUT	Indicates a command has timed out.
EWRPROTECT	Indicates an **open** operation requesting read/write mode was attempted on a read-only tape.
EWRPROTECT	Indicates that an **ioctl** operation that affects the media was attempted on a read-only tape.

Reliability and Serviceability Information

Errors returned from tape devices are categorized by the list described below:

GOOD COMPLETION	Indicates that the command completed successfully.
RECOVERED ERROR	Indicates that the command was successful after some recovery applied.
NOT READY	Indicates that the logical unit is offline.
MEDIUM ERROR	Indicates that the command terminated with a unrecovered media error condition. This may be caused by a tape flaw or a dirty head.
HARDWARE ERROR	Indicates that an unrecoverable hardware failure occurred during command execution or during a self test.
ILLEGAL REQUEST	Indicates that an illegal command or command parameter.
UNIT ATTENTION	Indicates the device has been reset or powered on.
DATA PROTECT	Indicates that a write was attempted on a write-protected tape.
BLANK CHECK	Indicates that a read command encountered a blank tape.
ABORTED COMMAND	Indicates the device aborted the command.

Medium, hardware, and aborted command errors from the above list are to be logged every time they occur. The ABORTED COMMAND error may be recoverable, but the error is logged if recovery fails. For the RECOVERED ERROR and recovered ABORTED COMMAND error types, thresholds are maintained and, when exceeded, an error is logged. These thresholds are then cleared.

Note: There are device-related adapter errors that are also logged every time they occur.

Error Record Values for Tape Device Media Errors

The fields defined in the error record template for tape device media errors are:

Comment	Equal to tape media error.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1332, indicating a tape operation failure.
Prob_Causes	Equal to 5003, indicating tape media.
User_Causes	Equal to 5100 and 7401, indicating a cause originating with the tape and defective media, respectively.
User_Actions	Equal to 1601 and 0000, indicating the removable media should be replaced and the operation retried and that problem determination procedures should be performed, respectively.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 5003, indicating tape media.
Fail_Actions	Equal to 1601 and 0000, indicating that the removable media should be replaced and the operation retried and that problem determination procedures should be performed, respectively.

The **detail_data** field contains the command type, device and adapter status, and the request sense information from the particular device in error. The **detail_data** field is contained in the **err_rec** structure. This structure is defined in the **<sys/errids.h>** file. The **sc_error_log_df** structure, which describes information contained in the **detail_data** field, is defined in the **<sys/scsi.h>** file.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request sense information.

Error Record Values for Tape or Hardware Aborted Command Errors

The fields in the **err_hdr** structure, as defined in the **<sys/erec.h>** header file for hardware errors and aborted command errors, are:

Comment	Equal to a tape hardware or aborted command error.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1331, indicating a tape drive failure.
Prob_Causes	Equal to 6314, indicating a tape drive error.
User_Causes	None.
User_Actions	Equal to 0000, indicating that problem determination procedures should be performed.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 5003 and 6314, indicating the failure cause is the tape and the tape drive, respectively.
Fail_Actions	Equal to 0000 indicating that problem determination procedures should be performed.

The **detail_data** field contains the command type, device and adapter status, and the request sense information from the particular device in error. The **detail_data** field is contained in the **err_rec** structure. This structure is defined in the **<sys/errids.h>** file. The **sc_error_log_df** structure, which describes information contained in the **detail_data** field, is defined in the **<sys/scsi.h>** file.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Error Record Values for Tape-Recovered Error Threshold Exceeded

The fields defined in the **err_hdr** structure, as defined in the **<sys/erec.h>** file for recovered errors that have exceeded the threshold counter, are:

Comment	Equal to tape recovered error threshold exceeded.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1331, indicating a tape drive failure.
Prob_Causes	Equal to 5003 and 6314, indicating the probable cause is the tape and tape drive, respectively.
User_Causes	Equal to 5100 and 7401, indicating that the media is defective and the read/write head is dirty, respectively.
User_Actions	Equal to 1601 and 0000, indicating that removable media should be replaced and the operation retried and that problem determination procedures should be performed, respectively.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 5003 and 6314, indicating the cause is the tape and tape drive, respectively.
Fail_Actions	Equal to 0000 indicating that problem determination procedures should be performed.

The **detail_data** field contains the command type, device and adapter status, and the request-sense information from the particular device in error. The field is contained in the **err_rec** structure. The **err_rec** structure is defined in the **<sys/errids.h>** field. The **detail_data** field also specifies the error type of the threshold exceeded. The **sc_error_log_df** structure, which describes information contained in the **detail_data** field, is defined in the **<sys/scsi.h>** field.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Error Record Values for Tape SCSI Adapter-Detected Errors

The fields in the **err_hdr** structure as defined in the **<sys/erec.h>** file for adapter-detected errors are:

Comment	Equal to a tape SCSI adapter detected error.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1331, indicating a tape drive failure.
Prob_Causes	Equal to 3300 and 6314, indicating an adapter and tape drive failure, respectively.
User_Causes	None.
User_Actions	Equal to 0000, indicating that problem determination procedures should be performed.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 3300 and 6314, indicating an adapter and tape drive failure, respectively.
Fail_Actions	Equal to 0000 indicating that problem determination procedures should be performed.

The **detail_data** field contains the command type and adapter status. This field is contained the **err_rec** structure which is defined by the **<sys/errids.h>** file. Request sense information is not available with this type of error. The **sc_error_log_df** structure describes information contained in the **detail_data** field and is defined in the **<sys/scsi.h>** file.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Error Record Values for Unknown Errors

Errors that should not occur are grouped in the unknown errors class. Data-protect errors fall into this class. These errors are detected by the tape device driver and should never be seen at the tape drive.

The **err_hdr** structure for unknown errors describes the following fields:

Comment	Equal to tape unknown error.
Class	Equal to all error classes.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to UNKN, indicating the error type is unknown.
Err_Desc	Equal to 0xFE00, indicating the error description is unknown.
Prob_Causes	None.
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 0xFFFF, indicating the failure cause is unknown.
Fail_Actions	Equal to 0000 indicating that problem determination procedures should be performed.

The **detail_data** field contains the command type and adapter status, and the request sense information from the particular device in error. The **detail_data** field is contained in the **err_rec** structure. This field is contained in the **<sys/errids.h>** field. The **sc_error_log_df** structure describes information contained in the **detail_data** field and is defined in the **<sys/scsi.h>** file.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Files

/dev/rmt0, /dev/rmt0.1, /dev/rmt0.2, ..., /dev/rmt0.7

/dev/rmt1, /dev/rmt1.1, /dev/rmt1.2, ..., /dev/rmt1.7..., /dev/rmt255, /dev/rmt255.1, /dev/rmt255.2, ..., /dev/rmt255.7

Related Information

The **rhdisk** special file, **rmt** special file.

The **open** subroutine, **close** subroutine, **read** subroutine, **write** subroutine, **ioctl** subroutine.

The SCSI Adapter Device Driver, The SCSI Device Driver-SCSI Adapter Device Driver Interface.

Special File Overview in *Files Reference*.

Device Driver Concepts Overview, SCSI Subsystem: Programming Introduction in *Kernel Extensions and Device Support Programming Concepts*.

Understanding Block I/O Device Drivers, Understanding I/O Access through Special Files, Understanding Pseudo-Devices in *Kernel Extensions and Device Support Programming Concepts*.

scdisk SCSI Device Driver

Purpose

Supports the SCSI physical volume (fixed-disk) device driver.

Syntax

#include <sys/devinfo.h>
#include <sys/scsi.h>
#include <sys/scdisk.h>

Description

Warning: Potential for data corruption or system crashes: Data corruption, loss of data, or loss of system integrity will occur if devices supporting paging, logical volumes, or mounted file systems are accessed using block special files. Block special files are provided for logical volumes and disk devices on AIX, and are solely for system use in managing file systems, paging devices and logical volumes. They should not be used for other purposes. Additional information concerning the use of special files may be obtained in Understanding I/O Access through Special Files.

The **<sys/scdisk.h>** special file provides raw I/O access and control functions to the physical disk device drivers on the RISC System/6000 machine platforms. The /**dev/hdisk** block special files are provided only for system use in managing file systems, paging devices and logical volumes. Raw I/O access is provided through the **/dev/rhdisk0**, **/dev/rhdisk1**, ..., character special files.

Due to performance considerations, direct access to physical disks through block special files should be avoided. In addition, direct access should be avoided to prevent data consistency problems that occur between data in the block I/O buffer cache and data in system pages.

Note: Configuration information for the hard-file is defined in the **<sys/scdisk.h>** file by the **disk_ddi** structure.

The prefix **r** on a special file name indicates that the drive is accessed as a raw device rather than a block device. Performing raw I/O with a fixed disk requires that all data transfers be in multiples of the disk block size. Also, all **lseek** subroutines that are made to the raw-disk device driver must result in a file pointer value that is a multiple of the disk block size.

Device-Dependent Subroutines

Typical fixed-disk operations are implemented using the **open, read, write,** and **close** subroutines.

The open and close Subroutines

The **openx** subroutine is intended primarily for use by the diagnostic commands and utilities. Appropriate authority is required for execution. Attempting to execute this subroutine without the proper authority results in a return value of –1, with the **errno** global variable set to EPERM.

The *ext* parameter passed to the **openx** subroutine selects the operation to be used for the target device. The **<sys/scsi.h>** file defines possible values for the *ext* parameter. The parameter can contain any combination of the following flag values logically ORed together:

- **SC_DIAGNOSTIC**

  Places the selected device in Diagnostic mode. This mode is singularly entrant. When a device is in Diagnostic mode, SCSI operations are performed during **open** or **close** operations, and error logging is disabled. In Diagnostic mode, only the **close** and **ioctl** operations are accepted. All other device-supported subroutines return a value of –1, with the **errno** global variable set to EACCES.

  A device can be opened in Diagnostic mode only if the target device is not currently opened. If an attempt is made to open a device in Diagnostic mode and the target device is already open, a value of –1 is returned and the **errno** global variable is set to EACCES.

- **SC_FORCED_OPEN**

  Forces a bus device reset (BDR) regardless of whether another initiator has the device reserved. The SCSI bus device reset is sent to the device before the **open** sequence begins, otherwise, the **open** operation executes normally.

- **SC_RETAIN_RESERVATION**

  Retains the reservation of the device after a **close** operation by not issuing the release. This flag prevents other initiators from using the device unless they break the host machine's reservation.

SCSI options to the **openx** subroutine gives more specific information on the open operations.

The readx and writex Subroutines

The **readx** and **writex** subroutines provide additional parameters affecting the raw data transfer. These subroutines pass the *ext* parameter which specifies request options. The options are constructed by logically OR-ing zero or more of the following values:

WRITEV	Indicates a request for write verification.
HWRELOC	Indicates a request for hardware relocation (safe relocation only).
UNSAFEREL	Indicates a request for unsafe hardware relocation.

The ioctl Subroutine

The IOCINFO operation is the only operation defined for all device drivers that use the **ioctl** subroutine. The remaining operations are all specific to the physical volume device.

Note: Diagnostic mode is required only for the DKIOCMD operation.

The following **ioctl** operations are available for physical volume devices:

- **IOCINFO**

  Returns the **devinfo** structure defined in the **<sys/devinfo.h>** file.

- **DKIORDSE**

  Provides a means for issuing a read command to the disk and obtaining the target device sense data on error. If the DKIORDSE operation returns a value of –1 and the **status_validity** field has **sc_valid_sense** set, then valid sense data is returned. Otherwise, target sense data is omitted.

  The DKIORDSE operation is provided for diagnostic use. It allows for the limited use of the target device while operating in an active system environment. The *arg* parameter to the DKIORDSE operation contains the address of a **sc_rdwrt** structure. This structure is defined in the **<sys/scsi.h>** file.

  The **devinfo** structure defines the maximum transfer size for a read. If an attempt is made to transfer more than the maximum, a value of –1 is returned and the **errno** global variable set to EINVAL. Refer to the Small Computer System Interface (SCSI) Specification for the applicable device for the particular request sense information.

- **DKIOWRSE**

  Provides a means for issuing a write command to the disk and obtaining the target device sense data on error. If the DKIOWRSE operation returns a value of –1 and the **status_validity** field has **sc_valid_sense** set, then valid sense data is returned. Otherwise, target-sense data is omitted.

  The DKIOWRSE operation is provided for diagnostic purposes to allow for limited use of the target device while operating in an active system environment. The *arg* parameter to the DKIOWRSE operation contains the address of a **sc_rdwrt** structure. This structure is defined in the **<sys/scsi.h>** file.

  The **devinfo** structure defines the maximum transfer size for a write. If an attempt is made to transfer more than the maximum, a value of –1 is returned and the **errno** global variable set to EINVAL. Refer to the Small Computer System Interface (SCSI) Specification for the applicable device for the particular request-sense information.

- **DKIOCMD**

  When the device has been successfully opened in the Diagnostic mode, this operation provides the means for issuing any SCSI command to the specified device. If the DKIOCMD is issued when the device is not in Diagnostic mode, a value of –1 is returned and the **errno** global variable set EACCES. The device driver performs no error recovery or logging on failures of this **ioctl** operation.

  The SCSI status byte and the adapter status bytes are returned via the *arg* parameter, which contains the address of a **sc_iocmd** structure (defined in the **<sys/scsi.h>** file). If the DKIOCMD operation returns a value of –1 and the **errno** global variable is set to a nonzero value, the requested operation has failed. In this case, the caller should evaluate the returned status bytes to determine why the operation failed and what recovery actions should be taken.

  The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum, a value of –1 is returned and the **errno** global variable set to EINVAL. Refer to the Small Computer System Interface (SCSI) Specification for the applicable device for the particular request-sense information.

Error Conditions

In addition to those errors listed, **ioctl, open, read,** and **write** subroutines against this device fail in the following circumstances:

EACCES	Indicates that an attempt was made to open a device currently opened in Diagnostic mode.
EACCES	Indicates that an attempt was made to open a diagnostic session on a device already opened.
EACCES	Indicates that the user attempted a subroutine other than an **ioctl** or **close** subroutine while in Diagnostic mode.
EACCES	Indicates that a DKIOCMD operation was attempted on a device not in Diagnostic mode.
EBUSY	Indicates that the target device is reserved by another initiator.
EINVAL	Indicates that the **read** or **write** subroutine supplied an *nbyte* parameter that is not an even multiple of the block size.
EINVAL	Indicates that a sense data buffer length of greater than 255 is invalid for a DKIOWRSE or DKIORDSE operation.
EINVAL	Indicates that the data buffer length exceeded the maximum defined in the **devinfo** structure for a DKIORSE, DKIOWRSE, or DKIOCMD **ioctl** operation.
EINVAL	Indicates that an unsupported **ioctl** operation was attempted.
EMEDIA	Indicates that the target device has indicated an unrecovered media error.
ENXIO	Indicates that the **ioctl** subroutine supplied an invalid parameter.
ENXIO	Indicates that a read or write command was attempted beyond the end of the disk.
EIO	Indicates that the target device cannot be located or is not responding.
EIO	Indicates that the target device has indicated an unrecovered hardware error.
EMFILE	Indicates that an **open** was attempted for an adapter which already has the maximum permissible number of opened devices.
EPERM	Indicates that the attempted subroutine requires appropriate authority.

Reliability and Serviceability Information

Errors returned from SCSI disk devices are categorized as follows:

GOOD COMPLETION	Indicates that the command completed successfully.
RECOVERED ERROR	Indicates that the command was successful after some recovery applied.
NOT READY	Indicates that the logical unit is offline.
MEDIUM ERROR	Indicates that the command terminated with a unrecovered media error condition.
HARDWARE ERROR	Indicates that an unrecoverable hardware failure occurred during command execution or during a self test.
ILLEGAL REQUEST	Indicates that an illegal command or command parameter.
UNIT ATTENTION	Indicates the device has been reset or powered on.
ABORTED COMMAND	Indicates the device aborted the command.
ADAPTER ERRORS	Indicates the adapter returned an error.

Error Record Values for Physical Volume Medium Errors

The fields defined in the error record for physical volume medium errors are:

Comment	Equal to physical volume media error.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1312, indicating a disk operation failure.
Prob_Causes	Equal to 5001, indicating DASD media.
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 5001, indicating DASD media.
Fail_Actions	Equal to 0000 to perform problem determination procedures.
Detail_Data	Equal to 156, 11 to indicate HEX.

The **detail_data** field in the **err_rec** structure contains the **sc_error_log_df** structure. The **err_rec** structure is defined in the **<sys/errid.h>** file. The **sc_error_log_df** structure is defined in the **<sys/scsi.h>** file.

The **sc_error_log_df** structure contains the following fields:

req_sense_data	Contains the request-sense information from the particular device that had the error.
reserved2	Contains the segment count, or number of megabytes read from the device at the time the error occurred.
reserved3	Contains the number of bytes read since the segment count was last incremented.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request sense information.

Error Record Values for Physical Volume Hardware Errors

The fields defined in the error record for physical volume hardware errors, and hard aborted command errors are:

Comment	Equal to physical volume hardware error.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1311, indicating a disk drive failure.
Prob_Causes	Equal to 6310, indicating a DASD drive.
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 6310 and 6330, indicating disk drive and disk drive electronics, respectively.
Fail_Actions	Equal to 0000 to perform problem determination procedures.
Detail_Data	Equal to 156, 11 to indicate HEX.

The **detail_data** field in the **err_rec** structure contains the **sc_error_log_df** structure. The **err_rec** structure is defined in the **<sys/errid.h>** file. The **sc_error_log_df** structure is defined in the **<sys/scsi.h>** file.

The **sc_error_log_df** structure contains the following fields:

req_sense_data	Contains the request-sense information from the particular device that had the error.
reserved2	Contains the segment count, or number of megabytes read from the device at the time the error occurred.
reserved3	Contains the number of bytes read since the segment count was last incremented.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Error Record Values for Adapter-Detected Physical Volume Hardware Failure

The fields defined in the error record for adapter-detected errors are:

Comment	Equal to adapter-detected physical volume hardware failure.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1311, indicating a disk drive failure.
Prob_Causes	Equal to 6310 and 3452, indicating a DASD drive, device cable cause, respectively.
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 6310, 6330, and 3452, indicating disk drive, disk drive electronics, and storage device cable as probable errors, respectively.
Fail_Actions	Equal to 0000 to perform problem determination procedures.
Detail_Data	Equal to 156, 11 to indicate HEX.

The **detail_data** field in the **err_rec** structure contains the **sc_error_log_df** structure. The **err_rec** structure is defined in the **<sys/errid.h>** file. The **sc_error_log_df** structure is defined in the **<sys/scsi.h>** file.

The **sc_error_log_df** structure contains the following fields:

reserved2	Contains the segment count, or number of megabytes read from the device at the time the error occurred.
reserved3	Contains the number of bytes read since the segment count was last incremented.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Error Record Values for Physical Volume Recovered Errors

The fields defined in the error record for recovered errors are:

Comment	Equal to physical volume recovered error.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to TEMP, indicating a temporary failure.
Err_Desc	Equal to 1611, indicating an impending storage subsystem failure.
Prob_Causes	Equal to 5001 and 6310, indicating a DASD media and DASD drive, respectively.
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 5001 and 6310, indicating DASD media and DASD device causes, respectively.
Fail_Actions	Equal to 0000 to perform problem determination procedures.
Detail_Data	Equal to 156, 11 to indicate HEX.

The **detail_data** field in the **err_rec** structure contains the **sc_error_log_df** structure. The **err_rec** structure is defined in the **<sys/errid.h>** file. The **sc_error_log_df** structure is defined in the **<sys/scsi.h>** file.

The **sc_error_log_df** structure contains the following fields:

req_sense_data	Contains the request sense information from the particular device that had the error, if it is valid.
reserved2	Contains the segment count, or number of megabytes read from the device at the time the error occurred.
reserved3	Contains the number of bytes read since the segment count was last incremented.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Error Record Values for Physical Volume Unknown Errors

The fields defined in the error record for unknown errors are:

Comment	Equal to physical volume hardware error.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to UNKN, indicating the type of error is unknown.
Err_Desc	Equal to FE00, indicating an undetermined error.
Prob_Causes	Equal to 6310, 5001, and 3300, indicating the DASD drive, DASD media, and the adapter, respectively.
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to FFFF, indicating the failure causes is unknown.
Fail_Actions	Equal to 2000, indicating the detail data should be reviewed.
Detail_Data	Equal to 156, 11 to indicate HEX.

The **detail_data** field in the **err_rec** structure contains the **sc_error_log_df** structure. The **err_rec** structure is defined in the **<sys/errid.h>** file. The **sc_error_log_df** structure is defined in the **<sys/scsi.h>** file.

The **sc_error_log_df** structure contains the following fields:

req_sense_data	Contains the request sense information from the particular device that had the error, if it is valid.
reserved2	Contains the segment count, or number of megabytes read from the device at the time the error occurred.
reserved3	Contains the number of bytes read since the segment count was last incremented.

Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the particular request-sense information.

Files

/dev/rhdisk0,/dev/rhdisk1,.....,/dev/rhdiskn

Related Information

The **rhdisk** special file.

The SCSI Adapter Device Driver, The SCSI Device Driver-SCSI Adapter Device Driver Interface.

Special File Overview in *Files Reference*.

Device Driver Concepts Overview, SCSI Subsystem Programming Introduction in *Kernel Extensions and Device Support Programming Concepts*.

Understanding Block I/O Device Drivers, Understanding I/O Access through Special Files, Understanding Pseudo-Devices in *Kernel Extensions and Device Support Programming Concepts*

SCSI Adapter Device Driver

Purpose

Supports the SCSI adapter.

Syntax

#include <sys/scsi.h>
#include <sys/devinfo.h>

Description

The **/dev/scsi***n* special file provides an interface to allow SCSI device drivers to access SCSI devices. It manages the adapter resources so that multiple SCSI device drivers can access devices on the same SCSI adapter simultaneously. SCSI adapters are accessed through the special files **/dev/scsi0, /dev/scsi1**, ...

Note: Configuration data for the adapter is defined in the device dependent structure (DDS). This structure is found in the **<sys/scsi.h>** file.

Device-Dependent Subroutines

The SCSI adapter device driver supports only the **open**, **close**, and **ioctl** subroutines. The **read** and **write** subroutines are not supported.

The open and close Subroutines

The **openx** subroutine provides an adapter diagnostic capability. The **openx** subroutine provides an *ext* parameter. This parameter selects the adapter mode and accepts the SC-DIAGNOSTIC value. This value is defined in the **<sys/scsi.h>** file and it places the device in Diagnostic mode.

In Diagnostic mode, only the **close** subroutine and **ioctl** operations are accepted. All other valid subroutines to the adapter return a value of –1 and the **errno** global variable is set to EACCES. In Diagnostic mode, the SCSI adapter device driver can accept the following requests:

- Run adapter diagnostics
- Run adapter wrap tests
- Download adapter microcode.

The **openx** subroutine requires appropriate authority to execute. Attempting to execute this subroutine without the proper authority returns a value of –1 and the **errno** global variable set to EPERM. Attempting to open a device already opened for normal operation or when another **openx** subroutine is in progress returns a value of –1 and the **errno** global variable set to EACCES.

Any kernel process can open the SCSI adapter device driver in Normal mode. For Normal mode, the *ext* parameter is set to 0 (zero). However, a non–kernel process must have at least appropriate authority to open the SCSI adapter device driver in Normal mode. Attempting to execute a normal **open** subroutine without the proper authority returns value of –1 and the **errno** global variable is set to EPERM.

ioctl Subroutine

Along with the IOCINFO operation, the SCSI device driver defines specific operations for devices in Non-diagnostic and Diagnostic mode.

The IOCINFO operation is defined for all device drivers that use the **ioctl** subroutine and is defined as follows:

- Returns a **devinfo** structure. This structure is defined in the **<sys/devinfo.h>** file. The device-type in this structure is DD_BUS, and the sub-type is DS_SCSI. The **flags** field is not used and is set to 0 (zero). Diagnostic mode is not required for this operation.
- The **devinfo** structure includes unique data such as the card SCSI ID, and the maximum data transfer size allowed (in bytes). A calling SCSI device driver uses this information to learn the maximum transfer size allowed for a device it controls on the SCSI adapter. In this way, the SCSI device driver can control devices across various SCSI adapters, with each device possibly having a different maximum transfer size.

SCSI ioctl Operations for Adapters in Non–Diagnostic mode

The Non-diagnostic operations are SCSI adapter device driver functions, rather than general device driver facilities. SCSI adapter device driver **ioctl** operations require that the adapter device driver is not in Diagnostic mode. If these operations are attempted while the adapter is in Diagnostic mode, a –1 is returned and the **errno** global variable is set to EACCES.

The following SCSI operations are for adapters in Non–diagnostic mode:

SCIOSTART	Opens a logical path to a SCSI device.
SCIOSTOP	Closes the logical path to a SCSI
SCIOINQU	Provides the means to issue an inquire command to a SCSI device.
SCIOSTUNIT	Provides the means to issue a SCSI Start Unit command to a selected SCSI adapter.
SCIOTUR	Allows a Test Unit Read command to the selected SCSI adapter.
SCIORESET	Allows the caller to force a SCSI device to release all current reservations, clear all current commands, and return to an initial state.
SCIOHALT	Aborts the current command (if there is one), clears the queue of any pending commands, and places the device queue in a halted state.

SCSI ioctl Operations for Adapters in Diagnostic Mode

The following operations for the **ioctl** subroutine are allowed only when the adapter has been successfully opened in Diagnostic mode. If these commands are attempted for an adapter not in Diagnostic mode, a value of –1 is returned and the **errno** global variable is set to EACCES.

SCIODIAG	Provides the means to issue adapter diagnostic commands
SCIOTRAM	Provides the means to issue various adapter commands to test the card DMA interface and buffer RAM.
SCIODNLD	Provides the means to download microcode to the adapter.

To allow these operations to be run on multiple SCSI adapter card interfaces, a special return value is defined. A return value of –1 with an **errno** value of ENXIO indicates that the requested **ioctl** is not applicable to the current adapter card. This return value should not be considered an error for commands which require Diagnostic mode for execution.

Summary of SCSI Error Conditions

Possible adapter device driver specific **errno** values are:

EACCES	Indicates that an **openx** subroutine was attempted while the adapter had one or more devices in use.
EACCES	Indicates that a subroutine other than the **ioctl** or **close** subroutine was attempted while the adapter was in Diagnostic mode.
EFAULT	Indicates that the adapter is indicating a diagnostic error in response to the SCIODIAG command. The SCIODIAG resume option must be issued to continue processing.
EFAULT	Indicates that a severe I/O error has occurred during an SCIODNLD command. Discontinue operations to this card.
EINVAL	Indicates that an invalid parameter. Data transfer length exceeds the adapter device driver's maximum transfer size.
EINVAL	Indicates that an invalid SCIOSTART parameters. This SCSI ID and LUN is already in use.
EINVAL	Indicates that an invalid SCIOSTART parameter. Device SCSI ID is the same as adapter SCSI ID.
EIO	Indicates that a delete operation has failed. The adapter is still open.
EIO	Indicates that an invalid **open** command. The adapter initialization must be executed before an **open** command is called.
EIO	Indicates that an invalid command. A SCIOSTART operation must be executed prior to this command, or an invalid SCSI ID and LUN combination must be passed in.
EIO	Indicates that the command has failed due to an error detected on the adapter or the SCSI bus.
EMFILE	Indicates that an SCIOSTART operation was attempted for an adapter that already has the maximum number of devices in use.
ENODEV	Indicates that the target device cannot be selected or is not responding.
ENXIO	Indicates that for diagnostic **ioctl** operations the requested command is not applicable to this adapter.
EPERM	Indicates that the caller did not have the required authority.
ETIMEDOUT	Indicates that a SCSI command or adapter command has exceeded the time-out value.

Reliability and Serviceability Information

Errors detected by the adapter device driver may be one of the following:

- Permanent adapter or system hardware errors
- Temporary adapter or system hardware errors
- Permanent unknown adapter microcode errors
- Temporary unknown adapter microcode errors
- Permanent unknown adapter device driver errors
- Temporary unknown adapter device driver errors

- Permanent unknown system errors
- Temporary unknown system errors

Permanent errors are either unretriable errors, or errors not recovered before a prescribed number of retries has been exhausted. Temporary errors are either non-retriable but non-catastrophic, or retriable and successfully recovered before a prescribed number of retries has been exhausted.

Error Record Values for Permanent Hardware Errors

The error record template for permanent hardware errors detected by the SCSI adapter device driver is described below. Refer to the **rc** structure for the actual definition of the detail data. The **rc** structure is defined in the **<sys/scsi.h>** file.

SCSI_ERR1:

Comment	Equal to permanent SCSI adapter hardware error.	
Class	Equal to H, indicating a hardware error.	
Report	Equal to TRUE, indicating this error should be included when an error report is generated.	
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.	
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.	
Err_Type	Equal to PERM, indicating a permanent failure.	
Err_Desc	Equal to 0x1010, indicating an adapter error.	
Prob_Causes	Equal to one of the following:	
	0x3300	Adapter hardware
	0x3400	Cable
	0x3461	Cable terminator
	0x6000	Device.
Fail_Causes	Equal to one of the following:	
	0x3300	Adapter
	0x3400	Cable loose or defective
	0x6000	Device.
Fail_Actions	Equal to one of the following:	
	0x000	Perform problem determination procedures.
	0x0301	Check the cable and its connections.
Detail_Data1	Equal to 108, 11, and HEX.	

Error Record Values for Temporary Hardware Errors

The error record template for temporary hardware errors detected by the SCSI adapter device driver follows:

SCSI_ERR2:

Comment	Equal to temporary SCSI adapter hardware error.	
Class	Equal to H, indicating a hardware error.	
Report	Equal to TRUE, indicating this error should be included when an error report is generated.	
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.	
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.	
Err_Type	Equal to TEMP, indicating a temporary failure.	
Err_Desc	Equal to 0x1010, indicating an adapter error.	
Prob_Causes	Equal to one of the following:	
	0x3300	Adapter hardware
	0x3400	Cable
	0x3461	Cable terminator
	0x6000	Device.
Fail_Causes	Equal to one of the following:	
	0x3300	Adapter
	0x3400	Cable loose or defective
	0x6000	Device.
Fail_Actions	Equal to one of the following:	
	0x000	Perform problem determination procedures.
	0x0301	Check the cable and its connections.
Detail_Data1	Equal to 108, 11, and HEX.	

Error Record Values for Permanent Unknown Adapter Microcode Errors

The error record template for permanent unknown SCSI adapter microcode errors detected by the SCSI adapter device driver follows:

SCSI_ERR3:

Comment	Equal to permanent SCSI adapter hardware error.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.

Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 0x1010, indicating an adapter error.
Prob_Causes	Equal to 0x3310 adapter microcode.
Fail_Causes	Equal to 0x3300 the adapter.
Fail_Actions	Equal to one of the following:

0x000	Perform problem determination procedures.
0x3301	If the problem persists then (0x3000) contact the appropriate service representatives.

Detail_Data1	Equal to 108, 11, and HEX.

Error Record Values for Temporary Unknown Adapter Microcode Errors

The error record template for temporary unknown SCSI adapter microcode errors detected by the SCSI adapter device driver follows:

SCSI_ERR4:

Comment	Equal to temporary unknown SCSI adapter sfw error.
Class	Equal to H.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to TEMP, indicating a temporary failure.
Err_Desc	Equal to 0x6100, indicating a microcode program error.
Prob_Causes	Equal to 3331, indicating adapter microcode.
Fail_Causes	Equal to 3330, indicating the adapter.
Fail_Actions	Equal to the following:

0x000	Perform problem determination procedures.
0x3301	If the problem persists then (0x3000) contact the appropriate service representatives.

Detail_Data1	Equal to 108, 11, and HEX.

Error Record Values for Permanent Unknown Adapter Device Driver Errors

The error record template for permanent unknown SCSI adapter device driver errors detected by the SCSI adapter device driver follows:

SCSI_ERR5:

Comment	Equal to permanent unknown driver error.
Class	Equal to S.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 0x2100, indicating a software program error.
Prob_Causes	Equal to 0X1000, indicating a software program.
Fail_Causes	Equal to 0X1000, indicating a software program.
Fail_Actions	Equal to 0x3301. If the problem persists then (0x3000) contact the appropriate service representatives.
Detail_Data1	Equal to 108, 11 and HEX.

Error Record Values for Temporary Unknown Adapter Device Driver Errors

The error record template for temporary unknown SCSI adapter device driver errors detected by the SCSI adapter device driver follows:

SCSI_ERR6

Comment	Equal to temporary unknown driver error.
Class	Equal to S.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to TEMP, indicating a temporary failure.
Err_Desc	Equal to 0x2100, indicating a software program error.
Prob_Causes	Equal to 0X1000, indicating a software program.
Fail_Causes	Equal to 0X1000, indicating a software program.
Fail_Actions	Equal to 0x3301. If the problem persists, (0x3000) contact the appropriate service representatives.
Detail_Data1	Equal to 108, 11, and HEX.

Error Record Values for Permanent Unknown System Errors

The error record template for permanent unknown system errors detected by the SCSI adapter device driver follows:

SCSI_ERR7:

Comment	Equal to permanent unknown system error.
Class	Equal to H.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to UNKN, indicating an unknown error.
Err_Desc	Equal to 0xFE00, indicating an undetermined error.
Prob_Causes	Equal to 0X1000, indicating a software program.
Fail_Causes	Equal to 0X1000, indicating a software program.
Fail_Actions	Equal to 0x000 and 0x3301, indicating perform problem determination procedures and if the problem persists then (0x3000) contact the appropriate service representatives.
Detail_Data1	Equal to 108, 11, and HEX.

Error Record Values for Temporary Unknown System Errors

The error record template for temporary unknown system errors detected by the SCSI adapter device driver follows:

SCSI_ERR8:

Comment	Equal to temporary unknown system error.
Class	Equal to H.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error should not be forwarded to the Network Alert Manager.
Err_Type	Equal to UNKN, indicating an unknown error.
Err_Desc	Equal to 0xFE00, indicating an undetermined error.
Prob_Causes	Equal to 0X1000, indicating a software program.
Fail_Causes	Equal to 0X1000, indicating a software program.
Fail_Actions	Equal to 0x000 and 0x3301, indicating perform problem determination procedures and if the problem persists then (0x3000) contact the appropriate service representatives.
Detail_Data1	Equal to 108, 11, and HEX.

Managing Dumps

The SCSI adapter device driver is a target for the system dump facility. The DUMPINIT and DUMPSTART options to the **dddump** entry point support multiple or redundant calls.

The DUMPQUERY option returns a minimum transfer size of 0 bytes and a maximum transfer size equal to the maximum transfer size supported by the SCSI adapter device driver.

To be processed, calls to the SCSI adapter device driver DUMPWRITE option should use the *arg* parameter as a pointer to the **sc_buf** structure. Using this interface, a SCSI write command may be executed on a previously started (opened) target device. The *uiop* parameter is ignored by the SCSI adapter device driver. Spanned, or consolidated, commands are not supported using DUMPWRITE.

Note: The various **sc_buf** status fields, including the **b_error** field, are not set at completion of the DUMPWRITE. Error logging is, of necessity, not supported during the dump.

Return Values

Successful completion of the **dddump** entry point is indicated by a 0 (zero). If unsuccessful, the entry point returns one of the following:

EINVAL	Indicates that the adapter device driver was passed an invalid request, such as attempting a DUMPSTART option before successfully executing a DUMPINIT option.
EIO	Indicates that the adapter device driver was unable to complete the command due to a lack of required resources or due to an I/O error.
ETIMEDOUT	Indicates that the adapter did not respond with status before the passed command time-out value expired.

Files

/dev/scsi0, **/dev/scsi1**, ...

Related Information

The SCSI **cdrom** device driver, SCSI **rmt** device driver, SCSI **scdisk** device driver.

The **scsi** special file.

The SCSI Device Driver-SCSI Adapter Device Driver Interface.

Special File Overview in *Files Reference.*

SCSI Subsystem Programming Introduction, Understanding I/O Access through Special Files in *Kernel Extensions and Device Support Programming Concepts*.

The following references are available for details about system power requirements:

- *Hardware Technical Reference — 7012 POWERstation and POWERserver.*
- *Hardware Technical Reference — 7013 and 7016 POWERstation and POWERserver.*
- *Hardware Technical Reference — 7015 POWERserver.*

SCIODIAG ioctl Operation

Purpose

Provides the means to issue adapter diagnostic commands.

Description

The SCIODIAG operation allows the caller to issue various adapter diagnostic commands to the selected SCSI adapter. These diagnostic command options are:

- Run the card Internal Diagnostics test
- Run the card SCSI Wrap test
- Run the card Read/Write Register test
- Run the card POS Register test
- Run the card SCSI Bus Reset test.

An additional option allows the caller to resume the card Internal Diagnostics test from the point of a failure, which is indicated by the return value. The *arg* parameter for the SCIODIAG operation is the address of a **sc_card_diag** structure. This structure is defined in the **<sys/scsi.h>** file.

The actual adapter error status information from each error reported by the card diagnostics is passed as returned parameters to the caller. Refer to the **sc_card_diag** structure defined in the **<sys/scsi.h>** header file for the format of the returned data.

When the card diagnostics have completed (with previous errors), a value of ENOMSG is returned. At this point, no further SCIODIAG resume commands are required, as the card internal diagnostics test has completed.

Adapter error status is always returned when a SCIODIAG operations results in an **errno** value of EFAULT. Because this error information is returned on each EFAULT, the final ENOMSG value returned for the card Internal Diagnostics test includes no error status information. Also, because this is a diagnostic command, these errors are not logged in the system error log.

Note: The SCSI adapter device driver performs no internal retries or other error-recovery procedures during execution of this operation. Error logging is also inhibited when running this command.

Return Values

When completed successfully this operation returns a value of 0 (zero). Otherwise, a value of –1 is returned and the **errno** global variable is set to one of the following:

ENXIO	Indicates that the operation or sub-option selected is not supported on this adapter. This should not be treated as an error. The caller must check for this return value first (before checking for other **errno** values) to avoid mistaking this for a failing command.
EFAULT	Indicates that the card internal diagnostics have detected an error and paused. To continue, the caller must issue another SCIODIAG operation with the resume option. In response to the resume option, the card continues the diagnostics until either the end is reached or another error is detected. The caller must continue to issue SCIODIAG operations until the EFAULT error no longer returns.
EIO	Indicates that the SCSI adapter device driver detected an error while attempting to run the SCIODIAG operation. In this case, the returned adapter status information must be analyzed to discover the cause of the error. Because this is a diagnostic command, this error is not logged in the system error log.
ETIMEDOUT	Indicates that the adapter did not respond with status before the passed command time-out value expired. The SCIODIAG operation is a diagnostic command and its errors are not logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**, ...

Related Information

The SCSI **cdrom** device driver, SCSI **scdisk** device driver, SCSI **rmt** device driver.

The SCSI Adapter Device Driver.

Special File Overview in *Files Reference*.

The SCSI Device Driver–SCSI Adapter Device Driver Interface, SCSI Subsystem Programming Introduction, Understanding I/O Access Through Special Files in *Kernel Extensions and Device Support Programming Concepts*.

SCIODNLD ioctl Operation

Purpose

Provides the means to download microcode to the adapter.

Description

The SCIODNLD operation provides for downloading microcode to the selected adapter. This operation can be used by system management routines to prepare the adapter for operation. The adapter must be opened in Diagnostic mode to ensure that no devices are in use when the SCIODNLD operation is executed.

There are two options for executing the SCIODNLD operation. The caller can either download microcode to the adapter or query the version of the currently downloaded microcode.

If the download microcode option is selected, a pointer to a download buffer and its length must be supplied in the caller's memory space. The maximum length of this microcode is adapter-dependent. If the adapter requires transfer of complete blocks, then the microcode to be sent must be padded to the next largest block boundary. The block size, if any, is adapter-dependent. Refer to the reference manual for the particular SCSI adapter to find the adapter-specific requirements of the microcode buffer to be downloaded.

The SCSI adapter device driver validates the parameter values for such things as maximum length and block boundaries as required. The *arg* parameter for the SCIODNLD operation is the address of a **sc_download** structure. This structure is defined in the **<sys/scsi.h>** file.

If the query version option is selected, the pointer and length fields in the passed parameter block are ignored. On successful completion of the SCIODNLD operation, the microcode version is contained in the **version_number** field.

The SCSI adapter device driver performs normal error recovery procedures during execution of the SCIODNLD operation.

Return Values

When completed sucessfully this operation returns a value of 0 (zero). Otherwise, a value of –1 is returned and the **errno** global variable is set to one of the following:

ENXIO	Indicates that the operation or sub-option selected is not supported on this adapter and should not be treated as an error. Note that the caller must check for this return value first (before checking for other **errno** values) to avoid mistaking this for a failing command.
EINVAL	Indicates that the adapter device driver was unable to execute the command due to invalid input parameters. Check microcode length and block boundary for errors.
EIO	Indicates that the adapter device driver was unable to complete the command due to an unrecoverable I/O error or microcode CRC error. If the card has on-board microcode, it may be able to continue running, and further commands may still be possible on this adapter. The adapter error status information is logged in the system error log.

EFAULT	Indicates that a severe I/O error has occurred, preventing completion of the download. In this case, further operations are not possible on the card, and the caller should discontinue commands to the card. The adapter error status information is logged in the system error log.
ETIMEDOUT	Indicates that the adapter did not respond with status before the passed command time-out value expired. Since the download operation may not have completed, further operations on the card may not be possible. The caller should discontinue sending commands to the card. This error is also logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**, ...

Related Information

The SCSI **cdrom** device driver, SCSI **rmt** device driver, SCSI **scdisk** device driver.

The SCSI Adapter Device Driver.

Special File Overview in *Files Reference.*

The SCSI Device Driver-SCSI Adapter Device Driver Interface, SCSI Subsystem Programming Introduction, Understanding I/O Access through Special Files in *Kernel Extensions and Device Support Programming Concepts.*

SCIOHALT ioctl Operation

Purpose

Aborts the current command (if there is one), clears the queue of any pending commands, and places the device queue in a halted state.

Description

The SCIOHALT operation allows the caller to abort the current command (if there is one) to a selected device, clear the queue of any pending commands, and place the device queue in a halted state. The command causes the attached SCSI adapter to execute a SCSI abort message to the selected target device. This command can be used by an upper-level SCSI device driver to abort a running operation, instead of waiting for the operation to complete or time out.

Once the SCIOHALT operation is sent, the calling device driver must set the **SC_RESUME** flag. This bit is located in the **flags** field of the next **sc_buf** structure to be processed by the SCSI adapter device driver. Any **sc_buf** structure sent without the **SC_RESUME** flag after the device queue is in the halted state is rejected.

The *arg* parameter to the SCIOHALT operation allows the caller to specify the SCSI identifier of the device to be reset. The least significant byte in the *arg* parameter is the LUN ID of the LUN on the SCSI controller to be halted. The next least significant byte is the SCSI ID. The remaining two bytes are reserved and must be set to zero.

The SCSI adapter device driver performs normal error recovery procedures during execution of this command. For example, if the abort message causes the SCSI bus to hang, a SCSI bus reset is initiated to clear the condition.

Return Values

When completed sucessfully this operation returns a value of 0 (zero). Otherwise, a value of –1 is returned and the **errno** global variable is set to one of the following:

EINVAL	Indicates that an SCIOSTART command was not issued prior to this command.
EIO	Indicates that an unrecoverable I/O error has occurred. In case of an unrecovered error, the adapter error status information is logged in the system error log.
ENODEV	Indicates that the target SCSI ID could not be selected or is not responding. This condition is not necessarily an error and is not logged.
ETIMEDOUT	Indicates that the adapter did not respond with status before the internal command time-out value expired. This error is logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**, ...

Related Information

The SCSI **cdrom** device driver, SCSI **rmt** device driver, SCSI **scdisk** device driver.

The SCSI Adapter Device Driver.

Special File Overview in *Files Reference.*

The SCSI Device Driver-SCSI Adapter Device Driver Interface, SCSI Subsystem Programming Introduction, Understanding I/O Access through Special Files in *Kernel Extensions and Device Support Programming Concepts.*

SCIOINQU ioctl Operation

Purpose

Provides the means to issue an inquiry command to a SCSI device.

Description

The SCIOINQU operation allows the caller to issue a SCSI device inquiry command to a selected adapter. This command can be used by system management routines to aid in configuration of SCSI devices.

The *arg* parameter for the SCIOINQU operation is the address of a **sc_inquiry** structure. This structure is defined in the **<sys/scsi.h>** file. The **sc_inquiry** parameter block allows the caller to select the SCSI and LUN IDs to be queried.

The **SC_ASYNC** flag byte of the parameter block must not be set on the initial call to this operation. This flag is only set if a bus fault occurs and the caller intends to attempt more than one retry.

If successful, the returned inquiry data can be found at the address specified by the caller in the **sc_inquiry** structure. Successful completion occurs if a device responds at the requested SCSI ID, but the returned inquiry data must be examined to see if the requested LUN exists. Refer to the Small Computer System Interface (SCSI) specification for the applicable device for the format of the returned data.

Note: The SCSI adapter device driver performs normal error-recovery procedures during execution of this command.

Return Values

When completed sucessfully this operation returns a value of 0 (zero). Otherwise, a value of –1 is returned and the **errno** global variable is set to one of the following:

EINVAL	Indicates that a SCIOSTART command was not issued prior to this command.
EIO	Indicates that an unrecoverable I/O error has occurred. If EIO is returned, the caller should retry the SCIOINQU operation since the first command may have cleared an error condition with the device. In case of an unrecovered error, the adapter error status information is logged in the system error log.
ENOCONNECT	Indicates that a bus fault has occurred. The caller should respond by retrying with the **SC_ASYNC** flag set in the flag byte of the passed parameters. If more than one retry is attempted, only the last retry should be made with the **SC_ASYNC** flag set. Since, in general, the SCSI adapter device driver cannot determine which device caused the SCSI bus fault, this error is not logged.
ENODEV	Indicates that no SCSI controller responded to the requested SCSI ID. This return value implies that no LUNs exist on the requested SCSI ID. Therefore, when the ENODEV return value is encountered, the caller can skip this SCSI ID (and all LUNs on it) and go on to the next SCSI ID. This condition is not necessarily an error and is not logged.
ETIMEDOUT	Indicates that the adapter did not respond with a status before the internal command time-out value expired. On receiving the ETIMEDOUT return value, the caller should retry this command at least once, since the first command may have cleared an error condition with the device. This error is logged in the system error log.

Files

/dev/scsi0, /dev/scsi1, ...

Related Information

The SCSI **cdrom** device driver, SCSI **rmt** device driver, SCSI **scdisk** device driver.

The SCSI Adapter Device Driver.

Special File Overview in *Files Reference*.

The SCSI Device Driver-SCSI Adapter Device Driver Interface, SCSI Subsystem Programming Introduction, Understanding I/O Access Through Special Files in *Kernel Extensions and Device Support Programming Concepts.*

SCIORESET ioctl Operation

Purpose

Allows the caller to force a SCSI device to release all current reservations, clear all current commands, and return to an initial state.

Description

The SCIORESET operation allows the caller to force a SCSI device to release all current reservations, clear all current commands, and return to an initial state. This command can be used by system management routines to force a SCSI controller to release a competing SCSI initiator's reservation in a multi-initiator environment.

The command actually executes a SCSI bus device reset (BDR) message to the selected SCSI controller on the selected adapter. The BDR message is directed at a SCSI ID. Therefore, all LUNs associated with that SCSI ID are affected by the execution of the BDR.

To take over a device effectively, a SCSI Reserve command should be issued after the SCUIRESET operation through the appropriate SCSI device driver. Typically, the SCSI device driver open logic issues a SCSI Reserve command. This prevents another initiator from claiming the device.

There is a finite amount of time between the release of all reservations (a SCIORESET operation) and the time the device is again reserved (a SCSI Reserve command from the host). During this interval, another SCSI initiator can reserve the device instead. If this occurs, the SCSI Reserve command from this host fails and the device remains reserved by a competing initiator. The intelligence needed to prevent or recover from this event is assumed to be beyond that of the SCSI adapter device driver and SCSI device driver components.

The *arg* parameter to the SCIORESET operation allows the caller to specify the SCSI identifier of the device to be reset. The least-significant byte in the *arg* parameter is the LUN ID of an LUN on the SCSI controller. The device indicated by the LUN ID should have been successfully started by a call to the SCIOSTART operation. The next-least-significant byte is the SCSI ID. The remaining two bytes are reserved and must be set to 0 (zero).

Examples of the Use of the SCIORESET Operation

The following example demonstrates actual use of this command. A SCSI ID of 1 is assumed, and a LUN of 0 exists on this SCSI controller.

```
open SCSI adapter device driver
SCIOSTART  SCSI ID=1,  LUN=0
SCIORESET SCSI ID=1,  LUN=0  (to free any reservations)
SCIOSTOP   SCSI ID=1,  LUN=0
close SCSI adapter device driver
open SCSI device driver (normal open) for SCSI ID=1,  LUN=0
...
Use device as normal
...
```

Another example makes use of the SC_FORCED_OPEN flag of the SCSI device driver:

```
open SCSI device driver (with SC_FORCED_OPEN flag) for SCSI ID=1,
LUN=0
...
```

Use device as normal.

The previous examples assume that the SCSI device driver **open** call executes a SCSI Reserve command on the selected device.

The SCSI adapter device driver performs normal error recovery procedures during execution of this command. For example, if the BDR message causes the SCSI bus to hang, a SCSI bus reset will be initiated to clear the condition.

Return Values

When completed sucessfully this operation returns a value of 0 (zero). Otherwise, a value of –1 is returned and the **errno** global variable is set to one of the following:

EINVAL	Indicates that a SCIOSTART command was not issued prior to this command.
EIO	Indicates that an unrecoverable I/O error has occurred. In case of an unrecovered error, the adapter error status information is logged in the system error log.
ENODEV	Indicates that the target SCSI ID could not be selected or is not responding. This condition is not necessarily an error and is not logged.
ETIMEDOUT	Indicates that the adapter did not respond with status before the internal command time-out value expired. This error is logged in the system error log.

Files

/dev/scsi0, /dev/scsi1, ...

Related Information

The SCSI **cdrom** device driver, SCSI **rmt** device driver, SCSI **scdisk** device driver.

The SCSI Adapter Device Driver.

Special File Overview in *Files Reference.*

The SCSI Device Driver-SCSI Adapter Device Driver Interface, SCSI Subsystem Programming Introduction, Understanding I/O Access Through Special Files in *Kernel Extensions and Device Support Programming Concepts.*

SCIOSTART ioctl Operation

Purpose

Opens a logical path to a SCSI device.

Description

The SCIOSTART operation opens a logical path to a SCSI device. This operation causes the adapter device driver to allocate and initialize the data areas needed to manage commands to a particular SCSI target.

The SCIOSTART operation must be issued prior to any of the other non-Diagnostic mode operations, such as SCIOINQU and SCIORESET. However, the SCIOSTART operation is not required prior to calling the IOCINFO operation. Finally, when the caller is finished issuing commands to the SCSI target, the SCIOSTOP operation must be issued to release allocated data areas and close the path to the device.

The *arg* parameter to SCIOSTART allows the caller to specify the SCSI and LUN identifier of the device to be started. The least significant byte in the *arg* parameter is the LUN, and the next least-significant byte is the SCSI ID. The remaining two bytes are reserved and must be set to 0 (zero).

Return Values

If completed successfully this operation returns a 0 (zero). Otherwise, a value of –1 is returned and the **errno** global variable set to one of the following values:

EIO	Indicates that sufficient system resources were lacking for completing the command.
EINVAL	Indicates either that the SCSI ID and LUN combination was invalid (the combination may already be in use) or that the passed SCSI ID is the same as that of the adapter.

If the SCIOSTART failed, the caller must not attempt other operations to this SCSI ID and LUN combination, since it is either already in use, or was never successfully started.

Files

/dev/scsi0, **/dev/scsi1**, ...

Related Information

The SCSI **cdrom** device driver, SCSI **rmt** device driver, SCSI **scdisk** device driver.

The SCSI Adapter Device Driver.

Special File Overview in *Files Reference.*

The SCSI Device Driver-SCSI Adapter Device Driver Interface, SCSI Subsystem Programming Introduction, Understanding I/O Access Through Special Files in *Kernel Extensions and Device Support Programming Concepts.*

SCIOSTOP ioctl Operation

Purpose

Closes the logical path to a SCSI

Description

The SCIOSTOP operation closes the logical path to a SCSI device. The SCIOSTOP causes the adapter device driver to deallocate data areas allocated in response to an SCIOSTART operation. This command must be issued when the caller wishes to cease communications to a particular SCSI target. The SCIOSTOP operation should only be issued for a device successfully opened by a previous call to an SCIOSTART operation.

The SCIOSTOP operation passes the *arg* parameter. This parameter allows the caller to specify the SCSI and LUN IDs of the device to be stopped. The least significant byte in the *arg* parameter is the LUN, and the next least significant byte is the SCSI ID. The remaining two bytes are reserved and must be set to zero.

Return Value

When this operation completes successfully a value of 0 (zero) is returned. Otherwise, a value of −1 is returned and the **errno** global variable is set to EIO. This code indicates that the device was not in the open state. An SCIOSTART operation should be issued prior to calling the SCIOSTOP operation.

Files

/dev/scsi0, **/dev/scsi1**, ...

Related Information

The SCSI **cdrom** device driver, SCSI **rmt** device driver, SCSI **scdisk** device driver.

The SCSI Adapter Device Driver.

Special File Overview in *Files Reference*.

The SCSI Device Driver-SCSI Adapter Device Driver Interface, SCSI Subsystem Programming Introduction, Understanding I/O Access Through Special Files in *Kernel Extensions and Device Support Programming Concepts.*

SCIOSTUNIT ioctl Operation

Purpose

Provides the means to issue a SCSI Start Unit command to a selected SCSI adapter.

Description

The SCIOSTUNIT operation allows the caller to issue a SCSI Start Unit command to a selected SCSI adapter. This command can be used by system management routines to aid in configuration of SCSI devices. For the SCIOSTUNIT operation, the *arg* parameter operation is the address of a **sc_startunit** structure. This structure is defined in the **<sys/scsi.h>** file.

The **sc_startunit** structure allows the caller to specify the SCSI and LUN IDs of the device on the SCSI adapter that is to be started. The **SC_ASYNC** flag (in the flag byte of the passed parameter block) must not be set on the initial attempt of this command.

The **start_flag** field in the parameter block allows the caller to indicate the start option to the SCIOSTUNIT operation. When **start_flag** is set to TRUE, the logical unit is to be made ready for use. When FALSE, the logical unit is to be stopped.

Warning: When the **immed_flag** field is set to TRUE, the SCSI adapter device driver allows simultaneous SCIOSTUNIT operations to any or all attached devices. It is important that when executing simultaneous SCSI Start Unit commands, the caller should allow a delay of at least 10 seconds between succeeding SCSI operations. The delay ensures that adequate power is available to devices sharing a common power supply. Failure to delay in this manner can cause damage to the system unit or to attached devices. Please consult the individual device technical specifications manuals for individual devices and the *IBM RISC System/6000 Technical Reference* for detailed power requirements.

The **immed_flag** field allows the caller to indicate the immediate option to the SCIOSTUNIT operation. When the **immed_flag** field is set to TRUE, status is to be returned as soon as the command is received by the device. When the field is set to FALSE, the status is to be returned after the operation is completed. The caller should set the **immed_flag** field to TRUE to allow overlapping SCIOSTUNIT operations to multiple devices on the SCSI bus. In this case, the SCIOTUR operation can be used to determine when the SCIOSTUNIT has actually completed.

Note: The SCSI adapter device driver performs normal error recovery procedures during execution of the SCIOSTUNIT operation.

Return Values

When completed successfully, the SCIOSTUNIT operation returns 0 (zero). Otherwise, a value of –1 is returned and the **errno** global variable is set to one of the following:

EINVAL	Indicates that an SCIOSTART command was not issued prior to this command.
EIO	Indicates that an unrecoverable I/O error has occurred. If EIO is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. In case of an unrecovered error, the adapter error status information is logged in the system error log.

ENOCONNECT	Indicates that a bus fault has occurred. The caller should respond by retrying with the **SC_ASYNC** flag set in the flag byte of the passed parameters. If more than one retry is attempted, only the last retry should be made with the **SC_ASYNC** flag set. Since, in general, the SCSI adapter device driver cannot determine which device caused the SCSI bus fault, this error is not logged.
ENODEV	Indicates that no SCSI controller responded to the requested SCSI ID. This condition is not necessarily an error and is not logged.
ETIMEDOUT	Indicates that the adapter did not respond with status before the internal command time-out value expired. If ETIMEDOUT is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. This error is logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**, ...

Related Information

The SCSI **cdrom** device driver, SCSI **rmt** device driver, SCSI **scdisk** device driver.

The SCSI Adapter Device Driver.

Special File Overview in *Files Reference.*

The SCSI Device Driver-SCSI Adapter Device Driver Interface, SCSI Subsystem Programming Introduction, Understanding I/O Access Through Special Files in *Kernel Extensions and Device Support Programming Concepts.*

SCIOTRAM ioctl Operation

Purpose

Provides the means to issue various adapter commands to test the card DMA interface and buffer RAM.

Description

The SCIOTRAM operation allows the caller to issue various adapter commands to test the card DMA interface and buffer RAM. The *arg* parameter block to the SCIOTRAM operation is the **sc_ram_test** structure. This structure is defined in the **<sys/scsi.h>** file and contains the following;

- A pointer to a read or write test pattern buffer
- The length of the buffer
- An option field indicating whether a read or write operation is requested.

Note: The SCSI adapter device driver is not responsible for comparing read data with previously written data. After successful completion of write or read operations, the caller is responsible for performing a comparison test to determine the final success or failure of this test.

The SCSI adapter device driver performs no internal retries or other error recovery procedures during execution of this operation. Error logging is inhibited when running this command.

Return Values

When completed sucessfully this operation returns a value of 0 (zero). Otherwise, a value of –1 is returned and the **errno** global variable is set to one of the following:

ENXIO	Indicates that the operation or sub-option selected is not supported on this adapter. This should not be treated as an error. The caller must check for this return value first (before other **errno** values) to avoid mistaking this for a failing command.
EIO	Indicates that the adapter device driver has detected an error. The specific adapter status is returned in the **sc_ram_test** parameter block. The SCIOTRAM operation is a diagnostic command and, as a result, this error is not logged in the system error log.
ETIMEDOUT	Indicates the adapter did not respond with status before the passed command time-out value expired. The SCIOTRAM operation is a diagnostic command and, as a result, this error is not logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**, ...

Related Information

The SCSI **cdrom** device driver, SCSI **scdisk** device driver, SCSI **rmt** device driver.

The SCSI Adapter Device Driver.

Special File Overview in *Files Reference*.

The SCSI Device Driver-SCSI Adapter Device Driver Interface, SCSI Subsystem Programming Introduction, Understanding I/O Access through Special Files in *Kernel Extensions and Device Support Programming Concepts*.

SCIOTUR ioctl Operation

Purpose

Allows a Test Unit Read command to the selected SCSI adapter.

Description

The SCIOTUR operation allows the caller to issue a SCSI Test Unit Read command to a selected SCSI adapter. This command can be used by system management routines to aid in configuration of SCSI devices. The *arg* parameter for the SCIOTUR operation is the address of a **sc_ready** structure. This structure is defined in the **<sys/scsi.h>** header file.

The **sc_ready** structure allows the caller to specify the SCSI and LUN ID of the device on the SCSI adapter that is to receive the SCIOTUR operation. The SC_ASYNC flag (in the flag byte of the *arg* parameter block) must not be set on the initial attempt of this command. The **sc_ready** structure provides two output fields, the **status_validity** and **scsi_status** field. Using these two fields, the SCIOTUR operation returns status is to the caller.

When an **errno** value of EIO is received, the caller should evaluate the returned status in the **status_validity** and **scsi_status** fields. The **status_validity** field is set to the value SC_SCSI_ERROR to indicate that the **scsi_status** field has a valid SCSI Bus status in it. The **<sys/scsi.h>** header file contains typical values for the **scsi_status** field.

Following an SCIOSTUNIT operation, a calling program can use the SCSI bus status to tell if the device is ready or not. If an **errno** value of EIO is returned, and the **status_validity** field is set to 0 (zero), then an unrecovered error occurred. If, on retry, the same result is obtained, the device should be skipped. If the **status_validity** field is set to SC_SCSI_ERROR, and the **scsi_status** field indicates a Check Condition status, then another SCIOTUR command should be sent after a reasonable delay (for example, several seconds).

After one or more attempts, the SCIOTUR operation should return a successful completion, indicating that the device was successfully started. If, after a reasonable number of seconds, the SCIOTUR operation still returns a **scsi_status** field set to a Check Condition status, the device should be skipped.

Note: The SCSI adapter device driver performs normal error recovery procedures during execution of this command.

Return Values

When completed sucessfully this operation returns a value of 0 (zero). For the SCIOTUR operation, this means the target device has been successfully started and is ready for data access. If unsuccessful, this operation returns a value of –1 and the **errno** global variable is set to one of the following:

EINVAL	Indicates that an SCIOSTART operation was not issued prior to this command.
EIO	Indicates that the adapter device driver was unable to complete the command due to an unrecoverable I/O error. If EIO is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. In case of an unrecovered error, the adapter error status information is logged in the system error log.

ENOCONNECT	Indicates that a bus fault has occurred. The caller should retry with the **SC_ASYNC** flag set in the flag byte of the passed parameters. If more than one retry is attempted, only the last retry should be made with the **SC_ASYNC** flag set. Since, in general, the SCSI adapter device driver cannot determine what device caused the SCSI bus fault, this error is not logged.
ENODEV	Indicates that no SCSI controller responded to the requested SCSI ID. This condition is not necessarily an error and is not logged.
ETIMEDOUT	Indicates that the adapter did not respond with a status before the internal command time-out value expired. If this return value is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. This error is logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**, ...

Related Information

The SCSI **cdrom** device driver, SCSI **rmt** device driver, SCSI **scdisk** device driver.

The SCSI Adapter Device Driver.

Special File Overview in *Files Reference.*

The SCSI Device Driver-SCSI Adapter Device Driver Interface, SCSI Subsystem Programming Introduction, Understanding I/O Access through Special Files in *Kernel Extensions and Device Support Programming Concepts.*

Index

Reader's Comment Form

AIX Calls and Subroutines Reference for IBM RISC System/6000

SC23–2198–00

Please use this form only to identify publication errors or to request changes in publications. Your comments assist us in improving our publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page	Comments

Please contact your IBM representative or your IBM–approved remarketer to request additional publications.

Please print

Date	________
Your Name	________
Company Name	________
Mailing Address	________
	________
	________
Phone No.	( ) ________ Area Code

No postage necessary if mailed in the U.S.A

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 997
11400 Burnet Rd.
Austin, Texas 78758–3493

Fold Fold

Cut or Fold Along Line

Fold and Tape Please Do Not Staple Fold and Tape