AIX Version 7.2

Technical Reference: Kernel and Subsystems, Volume 1

IBM

AIX Version 7.2

Technical Reference: Kernel and Subsystems, Volume 1

IBM

Note

Before using this information and the product it supports, read the information in “Notices” on page 693.

This edition applies to AIX Version 7.2 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this document

This topic collection is part of the six-volume technical reference set that provides information about system calls, kernel extension calls, and subroutines.

Highlighting

The following highlighting conventions are used in this document:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Bold highlighting also identifies graphical objects, such as buttons, labels, and icons that the you select.
Italics	Identifies parameters for actual names or values that you supply.
`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 text that you must type.

Case sensitivity in AIX

Everything in the AIX® operating system is case sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type `LS`, the system responds that the command `is not found`. Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

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

Technical Reference: Kernel and Subsystems, Volume 1

This topic collection provides information about kernel services, description of standard device driver entry points parameters, and the list of virtual file system operations.

The AIX operating system is designed to support The Open Group's Single UNIX Specification Version 3 (UNIX 03) for portability of operating systems based on the UNIX operating system. Many new interfaces, and some current ones, have been added or enhanced to meet this specification. To determine the correct way to develop a UNIX 03 portable application, see The Open Group's UNIX 03 specification on The UNIX System website (http://www.unix.org).

What's new in Technical Reference: Kernel and Subsystems, Volume 1

Read about new or significantly changed information for the Technical Reference: Kernel and Subsystems, Volume 1 topic collection.

How to see what's new or changed

In PDF files, you might see revision bars (|) in the left margin of new and changed information.

January 2017

Added the sleepx and in_localaddr kernel services.

December 2017

Added information about the **V_READMAKE** flag in the `vm_readp` kernel services.

May 2016

Added information about the `vsx_enable` and the `vsx_disable` kernel services.

December 2015

The following information is a summary of the updates made to this topic collection:

- Added information about the following kernel services:
  - “fp_get_path Kernel Service” on page 149
  - “fskv_reg Kernel Service” on page 174
  - “fskv_unreg Kernel Service” on page 178
  - “nameToXfid() Kernel Service” on page 376
  - “TE_verify_reg Kernel Service” on page 480
  - “TE_verify_unreg Kernel Service” on page 482
  - “xfidToName() Kernel Service” on page 590

Kernel Services

The following kernel services begin with the with the letter a - x.

a

The following kernel services begin with the with the letter a.

acct_add_LL or acct_zero_LL Kernel Service

Purpose

Increments counters for advanced accounting.

Syntax

```
unsigned long long acct_add_LL(ptr, incr)
unsigned long long *ptr;
unsigned int incr;

unsigned long long acct_zero_LL(ptr)
unsigned long long *ptr;
```

Parameters

Item	Description
ptr	Address of statistic to be incremented.
incr	Increment to be applied.

Description

These kernel services are special atomic increment and clear services that are designed to allow machine-independent updating of **unsigned long long** values. The increment service only performs an increment if advanced accounting is enabled.

The **acct_add_LL** kernel service adds the value associated with the *incr* parameter to the 64-bit counter at the address designated by the *ptr* parameter. The **acct_zero_LL** kernel service atomically zeroes the 64-bit counter.

Both routines return the previous value of the 64-bit counter. This way, the **acct_zero_LL** kernel service can be used to atomically get the most recent value and set the counter to NULL. Because only delta statistics are reported each interval, this capability is required by interval accounting when the accounting record is being built for a report.

Execution Environment

These kernel services can be called from either the interrupt environment or the process environment.

Return Values

These subroutines return the previous value of the location designated by the *ptr* parameter.

Related reference:

"acct_interval_register or acct_interval_unregister Kernel Service" on page 4

"acct_put Kernel Service" on page 6

acct_get_projid Kernel Service

Purpose

Gets the project identifier for the current process.

Syntax

```
projid_t acct_get_projid(void)
```

Description

The **acct_get_projid** kernel service returns the project identifier for the current process.

Execution Environment

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

Return Values

The **acct_get_projid** kernel service returns the current project identifier.

Related reference:

"acct_put Kernel Service" on page 6

acct_get_usage Kernel Service

Purpose

Allows kernel extensions to measure the resource utilization of transactions.

Syntax

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

unsigned long long acct_get_usage(usage)
struct tusage *usage;
```

Parameters

Item	Description
usage	Resource utilization structure.

Description

This routine is used to measure the resource utilization of a client transaction, so that the cost of the transaction can be included within the accounting record that identifies the client transaction. This accounting record is then used for chargeback purposes.

The **acct_get_usage** kernel service is designed to be called twice: once at the start of a transaction and a second time at the end of a transaction. Each time that the routine is called, it returns the resource utilization for the calling thread from creation using the *usage* parameter. Therefore, this routine can be called multiple times to determine the resource utilization of a code fragment by subtracting start and end values.

The following macros are provided for manipulating the usage parameter:

TUSAGE_ZERO(TU)
Initializes the **tusage** structure

TUSAGE_ADD(TU1, TU2)
Adds **tusage** structures (T1 = T1 + T2)

TUSAGE_SUB(TU1, TU2)
Subtracts **tusage** structures (T1 = T1 – T2)

The *usage* parameter provides thread-specific information, so the caller must ensure that this routine is called from the same thread context when measuring the utilization of a transaction. The return value identifies the calling thread context.

The **acct_get_usage** kernel service returns a token that identifies the calling context. This token can be logically compared with other tokens returned by this routine to ensure that start and stop invocations were made from the same thread. The scope of the token depends on the context of the calling program. If this routine is called under a pthread, then it returns a token representing the currently executing pthread. Otherwise, the **acct_get_usage** kernel service returns a token representing the currently executing kernel thread. In the former case, the token has process-wide scope; in the latter case, the token has system-wide scope.

Execution Environment

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

Return Values

Upon successful completion, the **acct_get_usage** kernel service returns a token that identifies the calling thread context.

Related reference:

"acct_get_projid Kernel Service" on page 2

"acct_interval_register or acct_interval_unregister Kernel Service"

"acct_put Kernel Service" on page 6

acct_interval_register or acct_interval_unregister Kernel Service

Purpose

Registers or unregisters an advanced accounting handler.

Syntax

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

int acct_interval_register(trid, cmds, handler, arg, reg_token, reg_name)
int trid;
int cmds;
int (*handler)(int trid, int cmds, void *arg);
void *arg;
unsigned long *reg_token;
char *reg_name;

int acct_interval_unregister(reg_token)
unsigned long reg_token;
```

Parameters

Item	Description
trid	Transaction identifier
cmds	Invocations supported by the advanced accounting handler
handler	Function descriptor for the handler
arg	Identifies the instance of the kernel extension
reg_token	Token that is returned to caller naming the instance of the registration
reg_name	Identifies the transaction using a string

Description

The **acct_interval_register** kernel service registers accounting records that are produced by the kernel extension with the advanced accounting subsystem. These accounting records are named through accounting transaction identifiers, which are provided by the caller. Transaction identifiers are persistent in nature, because they are used by report and analysis utilities to interpret transaction-specific accounting data. The transaction identifier is implicitly mapped to a template.

Transaction identifiers (and associated templates) used by AIX are defined in the **sys/aacct.h** file. Identifiers in the range of 0 – 127 are reserved for AIX. Vendors can choose any value in the range 128 – 256 for their accounting records. If two vendors choose the same value, report and analysis programs must reference other fields in the accounting record header to uniquely identify the source of the transaction; that way, they can apply the appropriate template. The *subproject* field (which specifies the command name of the logger) and *length* field can be used to identify the source of the transaction. Collisions are very unlikely to occur. The transaction identifier and the transaction name, which is provided by the *reg_name* field, are presented to the system administrator. Vendors should choose representative names for their transactions. The maximum length of a transaction name is 15 bytes.

Administrators can enable and disable transactions, and thereby drive callouts to the kernel extension. A function descriptor for the advanced accounting handler is provided through the *handler* parameter. The interface of this handler is:

```
int handler(int trid, int cmd, void *arg);
```

The *trid* parameter is the transaction being acted on. The *cmd* parameter describes the action. The *arg* parameter is a value that was specified at registration for this particular instance of the handler. The *arg* parameter is specific to the kernel extension.

The following *cmd* values are supported:

Item	Description
ACCT_CMD_ENABLE	The transaction is being enabled; start collecting.
ACCT_CMD_DISABLE	The transaction is being disabled; stop collecting.
ACCT_CMD_INTERVAL	The system interval has expired; provide accounting data.
ACCT_CMD_FSWITCH	The active accounting file has changed; provide meta data.

The handler is invoked in the process environment from a dedicated kernel-only thread that is part of the advanced accounting subsystem. The kernel extension registers for the callouts that should be made by logically ORing *cmd* values. The *cmds* parameter to the **acct_interval_register** kernel service is provided for this purpose.

When a transaction is enabled, the kernel extension should allocate accounting structures and start collecting statistics. When a transaction is disabled, the kernel extension should quit collecting statistics and free accounting structures. If a transaction is not enabled, the kernel subsystem should not collect statistics for the transaction. The kernel extension relies on the callout mechanism to provide notification when a transaction is enabled. This way, accounting records that are not required for the report are not collected and the accounting overhead is minimized.

If the kernel extension registers for interval accounting, the extension is called when the system interval expires. The handler should record its data using the **acct_put** kernel service and should reset its counters so that only delta statistics are produced in the next interval. The **acct_zero_LL** and **acct_add_LL** kernel services are provided so that statistics can be reported and zeroed atomically. When the system interval is disabled, the system automatically generates an interval callout to collect the last round of statistics.

The file switch callout is provided, so that subsystems can record accounting data in each accounting file. Most subsystems are not expected to use this option.

Execution Environment

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

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

Return Values

Upon successful completion, 0 is returned. If unsuccessful, **errno** is set to a value that explains the error.

Related reference:

"acct_add_LL or acct_zero_LL Kernel Service" on page 2

"acct_put Kernel Service"

acct_put Kernel Service

Purpose

Writes an accounting record.

Syntax

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

void acct_put(trid, flags, projid, usage, trdata, tr_len);
int trid;
int flags;
projid_t projid;
struct tusage *usage;
void *trdata;
int tr_len;
```

Parameters

Item **Description**

trid Transaction identifier.

flags Flags associated with the transaction or the production of the transaction. The following value is defined:

ACCT_PUT_DIRECT
Overrides aggregate transaction

projid Project identifier, associated with the transaction, that identifies the billable entity. The following values are defined:

PROJID_SYSTEM
This identifier is typically associated with system overhead and is often used for shared devices, such as disks and network adapters.

PROJID_UNKNOWN
This identifier is used when the billable entity is unknown to the caller. In this case, the system calculates the project identifier using the project assignment policy specified by the system administrator.

project identifier
If the project identifier is known, it should be specified.

usage Identifies the resource usage values associated with the transaction.

trdata Transaction-specific information.

tr_len Size of the transaction-specific data in bytes.

Description

The **acct_put** kernel service provides accounting data to the advanced accounting subsystem. This service builds the accounting record header from its parameters and values associated with the calling context. The transaction-specific data specified by the caller is copied after the header. This data is internally buffered so that it can be written efficiently to the accounting data file some time later.

The *trid* parameter identifies the type of transaction that is being provided and implicitly identifies the format of the transaction-specific data. This identifier is included within the accounting header and is used by report and analysis commands to infer the right template that can interpret transaction-specific data. Vendors are encouraged to document their transaction identifiers and record templates so that report and analysis tools can be produced to interpret this data.

Accounting transaction identifiers are defined in the following range:

Item	Description
0-127	AIX accounting transaction identifiers
128-255	Vendor accounting transaction identifiers

The **ACCT_PUT_DIRECT** flag is provided as an override to the aggregation of accounting records, which is an optional feature of the advanced accounting subsystem. By default, the system does not aggregate accounting data. Aggregation is designed to reduce the volume of data that is written to the accounting file. It is transparent to applications and middleware. When aggregation is enabled, the system throws out the transaction-specific data and produces statistics about the occurrence of the transaction and the aggregate resource utilization. The data is produced along project boundaries, so the ability to perform chargeback is not lost, although the data that is produced is different. Statistical information about the transaction is captured in the accounting file in lieu of the transaction.

Because aggregation might not be desirable in some cases, the **ACCT_PUT_DIRECT** flag is provided to override this feature. For example, because the significance of a transaction that describes the shared use of a disk is bound up in the transaction-specific data, the transaction cannot be effectively aggregated. The significance of the transaction is thrown out in the course of aggregation. In effect, the statistic has already been aggregated by the producer, so it should be written directly to the file instead of being aggregated again by the accounting subsystem.

The usage values pointed to by the *usage* parameter is calculated using the **acct_get_usage** kernel service. The *usage* parameter is optional. A value of NULL can be specified to signify no usage information. Aggregation uses this field to accumulate resource utilization. If this information is calculated for the transaction, it should be passed as a parameter to this routine, instead of just including it within the transaction-specific data section. The advanced accounting subsystem does not know the format of this section and cannot aggregate it. In such a case, this section would be thrown out when aggregation is enabled.

The *trdata* parameter contains the address of a buffer containing transaction-specific data, and the *tr_len* parameter identifies the number of bytes in this buffer that should be copied to the accounting file. A maximum of 16 KB of data can be written.

Execution Environment

The **acct_put** kernel service can be started from either the process or interrupt environment. However, aggregation of the transaction is only supported when the **acct_put** service is started from the process environment.

Return Values

The **acct_put** kernel service does not return a value.

Related reference:

"acct_add_LL or acct_zero_LL Kernel Service" on page 2

"acct_get_usage Kernel Service" on page 3

Related information:

acctctl Command

add_domain_af Kernel Service

Purpose

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

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/domain.h> int add_domain_af (*domain***) struct domain** **domain***;**

Parameter

Item	Description
domain	Specifies the domain of the address family.

Description

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

Execution Environment

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

Return Values

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

Example

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

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

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

Related reference:

"del_domain_af Kernel Service" on page 63

Related information:

Network Kernel Services

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

Item	Description
type	Specifies which type of protocol a packet contains. A value of x'FFFF' indicates that this input type is a wildcard type and matches all input packets.
service_level	Determines the processing level at which the protocol input handler is called. If the *service_level* parameter is set to **NET_OFF_LEVEL**, the input handler specified by the *isr* parameter is called directly. Setting the *service_level* parameter to **NET_KPROC** schedules a network dispatcher. This dispatcher calls the subroutine identified by the *isr* parameter.
isr	Identifies the routine that serves as the input handler for an input packet type.
ifq	Specifies an input queue for holding input buffers. If this parameter has a non-null value, an input buffer (**mbuf**) is enqueued. The *ifq* parameter must be specified if the processing level specified by the *service_level* parameter is **NET_KPROC.** Specifying null for this parameter generates a call to the input handler specified by the *isr* parameter, as in the following:
af	Specifies the address family of the calling protocol. The *af* parameter must be specified if the *ifq* parameter is not a null character. This parameter must be greater than or equal to 0 and less than **NETISR_MAX**. Refer to **netisr.h** for the range of values of *af* that are already in use. Also, other kernel extensions that are not AIX and that use network ISRs currently running on the system can make use of additional values not mentioned in **netisr.h**. `(*isr)(CommonPortion,Buffer);` In this example, `CommonPortion` points to the network common portion (the **arpcom** structure) of a network interface and `Buffer` is a pointer to a buffer (**mbuf**) containing an input packet.

Description

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

Execution Environment

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

Return Values

Item	Description
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.

Related reference:

"del_input_type Kernel Service" on page 64

"find_input_type Kernel Service" on page 142

Related information:

Network Kernel Services

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

Item	**Description**
soft_intr_level	Specifies the software interrupt level to add. This parameter must be greater than or equal to 0 and less than **NETISR_MAX**. Refer to **netisr.h** for the range of values of *soft_intr_level* that are already in use. Also, other kernel extensions that are not AIX and that use network ISRs currently running on the system can make use of additional values not mentioned in **netisr.h**.
service_level	Specifies the processing level of the network software interrupt.
isr	Specifies the interrupt service routine to add.

Description

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

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

Execution Environment

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

Return Values

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

Related reference:

"del_netisr Kernel Service" on page 65

Related information:

Network Kernel Services

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

Item	Description
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.

Related reference:

"del_netopt Macro" on page 66

Related information:

no Command

Network Kernel Services

as_att64 Kernel Service

Purpose

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

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/vmuser.h> #include <sys/adspace.h> unsigned long long as_att64 (*vmhandle*, *offset***) vmhandle_t** *vmhandle***; int** *offset***;**

Parameters

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

Description

Item	Description
The **as_att64** kernel service:	Selects an unallocated region within the current user address space. Allocates the region. Maps the virtual memory object selected by the vmhandle parameter with the access permission specified in the handle. Constructs the address of the offset specified by the offset parameter within the user-address space.

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

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

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

Execution Environment

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

Return Values

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

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

Related reference:

"as_seth64 Kernel Service" on page 21

"as_geth64 Kernel Service" on page 14

"as_getsrval64 Kernel Service" on page 15

as_det64 Kernel Service

Purpose

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

Syntax

#include <sys/errno.h> #include <sys/adspace.h> int as_det64 (*addr64***) unsigned long long** *addr64*;

Parameters

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

Description

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

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

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

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

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

Execution Environment

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

Return Values

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

Related reference:

"as_seth64 Kernel Service" on page 21

"as_geth64 Kernel Service" on page 14

"as_getsrval64 Kernel Service" on page 15

as_geth Kernel Service

Purpose

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

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/vmuser.h> #include <sys/adspace.h> vmhandle_t as_geth (*Adspacep*, *Addr***) adspace_t ***Adspacep*; **caddr_t** *Addr*;

Parameters

Item	Description
Adspacep	Points to the address space structure to obtain the virtual memory object handle from.
Addr	Specifies the virtual memory address that should be used to determine the virtual memory object handle for the specified address space.

Description

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

This service can also be called from the interrupt environment.

Execution Environment

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

Return Values

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

Related reference:

"vm_att Kernel Service" on page 535

as_geth64 Kernel Service

Purpose

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

Syntax

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

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

Parameter

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

Description

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

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

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

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

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

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

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

Execution Environment

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

Return Values

On successful completion, this routine returns the appropriate handle.

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

Errors include: Input address out of range.

Related reference:

"as_seth64 Kernel Service" on page 21

"as_getsrval64 Kernel Service"

"as_puth64 Kernel Service" on page 20

as_getsrval64 Kernel Service

Purpose

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

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/vmuser.h> #include <sys/adspace.h> vmhandle_t as_getsrval64 (*addr64***) unsigned long long** *addr64***;**

Parameters

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

Description

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

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

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

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

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

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

Execution Environment

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

Return Values

On successful completion this routine returns the appropriate handle.

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

Errors include: Input address out of range.

Related reference:

“as_geth64 Kernel Service” on page 14

“as_puth64 Kernel Service” on page 20

“as_seth64 Kernel Service” on page 21

as_lw_att64 Kernel Service

Purpose

Allocates and maps a specified region in the current user address space. Part of the lightweight kernel service subsystem, which must be initialized with the **as_lw_pool_init** kernel service before it can be used.

Syntax

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

int as_lw_att64 (dp, offset, length, addr)
xmem* dp;
size_t offset;
size_t length;
ptr64* addr;
```

Parameters

Item	Description
dp	Pointer to a cross memory descriptor that describes the virtual memory object that is being made addressable in the address space.
offset	Specifies the byte offset in the virtual memory object.
length	Specifies the number of bytes to map in the virtual memory object.
addr	Pointer to the location where the address will be returned.

Description

The **as_lw_att64** kernel service does the following:

- Allocates a region from the process' address space for the mapping.
- Maps the virtual memory object selected by the *dp* parameter.
- Constructs the address of the offset specified by the *offset* parameter within the user-address space.

Note: The **as_lw_att64** kernel service should be used with caution. Be sure to read the documentation for this and the other lightweight services (**as_lw_det64** and **as_lw_pool_init**) carefully before doing so. There is a risk of illegal data access and cross-process data corruption if these services are not used correctly.

In order to use this service, the cross memory descriptor pointed to by the *dp* parameter must be initialized by using the **xmattach** kernel service with the **LW_XMATTACH** flag set. The **lw_pool_init** kernel service must also have been successfully called by the current process.

The service will map an area length *bytes* long into the caller's address space from the memory represented by the descriptor, starting at the number of bytes specified in the *offset* parameter. It is illegal for any thread other than the caller of this service to address the attached region.

This service will operate correctly only in 64-bit user address spaces. It will not work for kernel processes (kprocs).

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

Execution Environment

The **as_lw_att64** kernel service can be called from the process environment only.

Return Values

On successful completion, this service sets the value of *addr* to the address of the allocated region and returns `0`.

Item	Description
NULL	An error occurred and errno indicates the cause.
EINVAL	Cross memory descriptor is in an invalid state, length is zero or offset plus length goes past the end of the virtual memory object.
ENODEV	The **as_lw_pool_init** kernel service has not been called to initialize the pool settings for this process.
ENOSYS	Called by a 32-bit process.
ENOSPC	Resources allocated to do lightweight services for this thread expended. Either the region to be attached is too large (the **as_lw_pool_init** kernel service was called with too small a *pool_size*) or there are outstanding attaches which need to release their lightweight resources using the **as_lw_det64** kernel service before this attach can be completed.
EIO	Indicates a failure of the lightweight subsystem, process should discontinue use of lightweight kernel services.
EPERM	Called by a user thread that is not 1:1 with a kernel thread.
ENOMEM	Could not allocate system resources for lightweight services for this thread.

Implementation Specifics

The **as_lw_att64** kernel service is part of Base Operating System (BOS) Runtime.

Related reference:

"as_lw_det64 Kernel Service" on page 18

“as_lw_pool_init Kernel Service” on page 19

as_lw_det64 Kernel Service

Purpose

Unmaps and deallocates a region in the current user address space that was mapped using the **as_lw_att64** kernel service.

Syntax

```
#include <sys/errno.h>
#include <sys/adspace.h>
#include <sys/xmem.h>
int as_lw_det64 (dp, addr, length)
xmem* dp;
void* addr;
size_t length;
```

Parameters

Item	Description
dp	The cross memory descriptor describing the attached virtual memory.
addr	Specifies the first effective address of the region to be deallocated.
length	Specifies the length of the region to be deallocated.

Description

Note: The **as_lw_det64** kernel service should be used with caution. Read the documentation for this and the other lightweight services (**as_lw_att64** and **as_lw_pool_init**) carefully before doing so. There is a risk that illegal data accesses will be allowed if these services are not used correctly.

The **as_lw_det64** kernel service unmaps the virtual memory from the region starting at the specified effective address, which is specified by the *addr* parameter. This service (and only this service) must be used to unmap regions mapped by the **as_lw_att64** kernel service. It must be called by the same thread that called the **as_lw_att64** kernel service. The *addr* parameter must be the value returned by the **as_lw_att64** kernel service, and the *dp* parameter and the *length* parameter must be the same *dp* and *length* passed to it. The *xmdetach* kernel service must not be called to release the *dp* parameter until any outstanding attaches of the *dp* parameter using the **as_lw_att64** kernel service have been detached using the **as_lw_det64** kernel service.

The **as_lw_det64** kernel service cannot be used to detach a region not mapped by the **as_lw_att64** kernel service.

The **as_lw_det64** kernel service will operate correctly only for 64-bit user address spaces. It will not work for kernel processes (kprocs).

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

Execution Environment

The **as_lw_det64** kernel service can be called from the process environment only.

Return Values

Item	Description
0	The region was successfully unmapped and deallocated.
EINVAL	An attempt was made to deallocate a region that should not have been deallocated.
ENOSYS	The service was called by a 32-bit process.
ENOMEM	No lightweight resources allocated to this thread.
EIO	Indicates a failure of the lightweight subsystem, process should discontinue use of lightweight kernel services.
EPERM	Called by a user thread that is not 1:1 with a kernel thread.

Implementation Specifics

The **as_lw_det64** kernel service is part of Base Operating System (BOS) Runtime.

Related reference:

"as_lw_att64 Kernel Service" on page 16

"as_lw_pool_init Kernel Service"

as_lw_pool_init Kernel Service

Purpose

Initializes lightweight attach and detach subsystem for the current process with the given settings.

Syntax

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

int as_lw_pool_init (pool_size, flags)
size_t pool_size;
uint flags;
```

Parameters

Item	Description
pool_size	Specifies the maximum number of bytes that can be attached by lightweight services at one time by each thread of this process.
flags	Specifies flag options for this kernel service. Valid values are `0` and `LW_DEBUG`.

Description

Note: The **as_lw_pool_init** kernel service should be used with caution. Read the documentation for this and the other lightweight services (**as_lw_att64** and **as_lw_det64**) carefully before doing so. There is a risk that illegal data accesses will be allowed if these services are not used correctly.

The **as_lw_pool_init** kernel service initializes the lightweight pool size and flag settings for the current process. Once it has been called, these settings are fixed and cannot be changed for the process.

If **LW_DEBUG** is set in the *flags* parameter, the risk of illegal data access will be removed from calls to the **as_lw_att64** kernel service and the **as_lw_det64** kernel service. This setting allows users to debug problems that are caused by incorrect use of these services.

Processes that have called the **as_lw_pool_init** kernel service can use the other lightweight kernel services (**as_lw_att64** and **as_lw_det64**) to attach and detach virtual memory regions represented by a cross memory descriptor. These kernel services are used on a per-thread basis, that is if one thread uses the

as_lw_att64 kernel service to attach virtual memory to a region of its address space, that region cannot be addressed by any other thread, and it must be detached by the same thread by using the **as_lw_det64** kernel service.

This service will operate correctly only for 64-bit user address spaces. It will not work for kernel processes (kprocs).

Execution Environment

The **as_lw_pool_init** kernel service can be called from a 64-bit process environment only.

Return Values

On successful completion, this service returns 0.

Item	Description
ENOSYS	The service was called by a 32-bit process.
EEXIST	The **as_lw_pool_init** kernel service has already been successfully completed for this process.
EINVAL	Invalid flag settings or the *pool_size* parameter is `0`.
EPERM	Called by a user thread that is not 1:1 with a kernel thread.

Implementation Specifics

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

Related reference:

"as_lw_att64 Kernel Service" on page 16

"as_lw_det64 Kernel Service" on page 18

as_puth64 Kernel Service

Purpose

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

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/vmuser.h> #include <sys/adspace.h> int as_puth64 (*addr64*, *vmhandle* **) unsigned long long** *addr64***; vmhandle_t** *vmhandle***;**

Parameters

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

Description

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

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

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

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

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

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

Execution Environment

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

Return Values

Item	Description
0	Successful completion.
EINVAL	Input address out of range.

Related reference:
"as_getsrval64 Kernel Service" on page 15
"as_geth64 Kernel Service" on page 14
"as_seth64 Kernel Service"

as_seth64 Kernel Service

Purpose

Maps a specified region for the specified virtual memory object.

Syntax

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

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

Parameters

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

Description

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

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

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

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

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

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

Execution Environment

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

Return Values

Item	Description
0	Successful completion.
EINVAL	Input address out of range.

Related reference:

"as_det64 Kernel Service" on page 12

"as_geth64 Kernel Service" on page 14

"as_puth64 Kernel Service" on page 20

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

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

Description

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

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

Execution Environment

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

Return Values

Item	Description
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.

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

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

Description

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

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

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

Execution Environment

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

Return Values

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

Related reference:

"audit_svcfinis Kernel Service"

"audit_svcstart Kernel Service"

Related information:

Security Kernel Services

audit_svcfinis Kernel Service

Purpose

Writes an audit record for a kernel service.

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/audit.h> int audit_svcfinis ( )

Description

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

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

Execution Environment

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

Return Values

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

Related reference:

"audit_svcbcopy Kernel Service" on page 23

"audit_svcstart Kernel Service"

Related information:

Security Kernel Services

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

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

Description

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

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

Execution Environment

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

Return Values

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

Example

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

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

Related reference:

"audit_svcbcopy Kernel Service" on page 23

"audit_svcfinis Kernel Service" on page 24

Related information:

Security Kernel Services

b

The following kernel services begin with the with the letter b.

bawrite Kernel Service

Purpose

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

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/buf.h> int bawrite (*bp***) struct buf ****bp***;**

Parameter

Item	Description
bp	Specifies the address of the buffer structure. On a platform that supports storage keys, the passed in *bp* parameter must be in the **KKEY_PUBLIC** or **KKEY_BLOCK_DEV** protection domain.

Description

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

Execution Environment

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

Return Values

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

Related reference:

"bwrite Kernel Service" on page 36

Related information:

Block I/O buffer cache kernel services overview

I/O Kernel Services

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

Item	Description
bp	Specifies the address of the buffer structure for the buffer to be written. On a platform that supports storage keys, the passed in *bp* parameter must be in the **KKEY_PUBLIC** or **KKEY_BLOCK_DEV** protection domain.

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.

Execution Environment

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

Return Values

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

Related reference:

"brelse Kernel Service" on page 32

Related information:

Block I/O Buffer Cache Kernel Services: Overview

I/O Kernel Services

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

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

Description

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

Execution Environment

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

Related reference:

"bwrite Kernel Service" on page 36

Related information:

Block I/O Buffer Cache Kernel Services: Overview

I/O Kernel Services

bindprocessor Kernel Service

Purpose

Binds or unbinds kernel threads to a processor.

Syntax

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

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

Parameters

Item	Description
What	Specifies whether a process or a kernel thread is being bound to a processor. The *What* parameter can take one of the following values: **BINDPROCESS** A process is being bound to a processor. **BINDTHREAD** A kernel thread is being bound to a processor.
Who	Indicates a process or kernel thread identifier, as appropriate for the *What* parameter, specifying the process or kernel thread which is to be bound to a processor.
Where	If the *Where* parameter is in the range 0-*n* (where *n* is the number of online processors in the system), it represents a bind CPU identifier to which the process or kernel thread is to be bound. Otherwise, it represents a processor class, from which a processor will be selected. A value of **PROCESSOR_CLASS_ANY** unbinds the specified process or kernel thread, which will then be able to run on any processor.

Description

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

Programs that use processor bindings must be aware of Dynamic Logical Partitioning (DLPAR).

Return Values

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

Error Codes

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

Item	Description
EINVAL	The *What* parameter is invalid, or the *Where* parameter indicates an invalid processor number or a processor class which is not currently available.
ESRCH	The specified process or thread does not exist.
EPERM	The caller does not have root user authority, and the *Who* parameter specifies either a process, or a thread belonging to a process, having a real or effective user ID different from that of the calling process.

Execution Environment

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

Related information:

bindprocessor command

fork subroutine

sysconf subroutine

Dynamic Logical Partitioning

binval Kernel Service

Purpose

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

Syntax

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

void binval ( dev)
dev_t  dev;
```

Parameter

Item	Description
dev	Specifies the device to be purged.

Description

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

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

Execution Environment

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

Return Values

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

Related reference:

“bflush Kernel Service” on page 27

Related information:

Block I/O Buffer Cache Kernel Services Overview

I/O Kernel Services

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

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

Description

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

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

Execution Environment

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

Return Values

Item	Description
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.

Related reference:

"bwrite Kernel Service" on page 36

Related information:

Block I/O Buffer Cache Kernel Services: Overview

I/O Kernel Services

bread Kernel Service

Purpose

Reads the specified block data into a buffer.

Syntax

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

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

Parameters

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

Description

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

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

Execution Environment

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

Return Values

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

On a platform that supports storage keys, the buffer header is allocated from the storage protected by the **KKEY_BLOCK_DEV** kernel key.

Related reference:

"iowait Kernel Service" on page 232

Related information:

Block I/O Buffer Cache Kernel Services: Overview

I/O Kernel Services

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

Item	Description
dev	Specifies the device containing the block to be read.
blkno	Specifies the block to be read.
rablkno	Specifies the read-ahead block to be read.

Description

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

- Obtain the block.
- Return the buffer header.

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

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

Execution Environment

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

Return Values

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

On a platform that supports storage keys, the buffer header is allocated from the storage protected by the **KKEY_BLOCK_DEV** kernel key.

Related reference:

"bread Kernel Service" on page 30

"iowait Kernel Service" on page 232

Related information:

Block I/O Buffer Cache Kernel Services: Overview

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

Item	Description
bp	Specifies the address of the **buf** structure to be freed. On a platform that supports storage keys, the passed in *bp* parameter must be in the **KKEY_PUBLIC** or **KKEY_BLOCK_DEV** protection domain.

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.

Related reference:

"geteblk Kernel Service" on page 185

"buf Structure" on page 614

Related information:

I/O Kernel Services

bsr_alloc Kernel Service

Purpose

Allocates a Barrier Synchronization Register (BSR) resource, and retrieves mapping information.

Syntax

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

int bsr_alloc (
 int bsr_bytes,
 struct io_map * bsr_map,
 int *bsr_stride,
 int *bsr_id)
```

Parameters

Item	Description
bsr_bytes	Number of BSR bytes wanted.
bsr_map	Mapping information for the BSR facility
bsr_stride	Stride at which the BSR bytes repeat within the mapping
bsr_id	An opaque identifier for the allocated BSR resource

Description

The **bsr_alloc** service can be used to allocate and reserve all or a portion of the BSR facility. The requested number of BSR bytes to allocate is communicated through the *bsr_bytes* parameter. The requested number of bytes must correspond to a supported window size, as communicated by the *supported_window_mask* parameter of the **bsr_query** service. If the requested number of bytes is available, the bytes are reserved and the I/O mapping information for accessing the allocated facility is written to the *bsr_map* structure. In addition, the stride within the mapping that the allocated BSR bytes repeat is recorded in the *bsr_stride* field. The *bsr_id* field is written with a unique identifier to be used with the **bsr_free** call.

If multiple granules or windows are to be used, they must be allocated with independent calls to **bsr_alloc**. this is because I/O mappings for multiple granules might not be contiguous, and strides are only applicable within the granule.

The resulting *bsr_map* information can then be used as input to **rmmap_create** for establishing addressability to the BSR resource within the current process address space.

Execution Environment

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

Return Values

If successful, **bsr_alloc** returns `0` and modifies the *bsr_map* structure so that it contains the mapping information for the newly allocated resource, modifies the *bsr_stride* field displays the stride on which the BSR bytes repeat within the mapping, and modifies the *bsr_id* field so that it displays a unique identifier for the newly allocated BSR resource. If unsuccessful, one of the following values is returned:

Item	Description
ENODEV	The BSR facility does not exist.
EINVAL	Unsupported number of bytes requested.
EBUSY	Requested BSR bytes or mappable BSR windows are currently in use.

Related reference:

"bsr_free Kernel Service"

"bsr_query Kernel Service" on page 35

"rmmap_create Kernel Service" on page 449

bsr_free Kernel Service

Purpose

Frees a Barrier Synchronization Register (BSR) resource previously allocated with the **bsr_alloc** kernel service.

Syntax

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

int bsr_free (
 int bsr_id,
```

Parameters

Item	Description
bsr_id	BSR resource identifier as returned in the *bsr_id* field of the **bsr_alloc** call.

Description

The **bsr_free** service releases a BSR allocation. The specific BSR resource being freed is identified by the unique identifier *bsr_id* from the corresponding **bsr_alloc** call.

It is the caller's responsibility to ensure that all prior attachments to the BSR resource, through **rmmap_create** calls, have been detached with corresponding **rmmap_remove** calls prior to freeing the BSR resource.

Execution Environment

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

Return Values

Item	Description
0	A successful operation.
ENODEV	The BSR facility is not present.
EINVAL	BSR resource corresponding to *bsr_id* is invalid or not currently allocated.

Related reference:

"bsr_alloc Kernel Service" on page 33

"bsr_query Kernel Service"

"rmmap_remove Kernel Service" on page 453

bsr_query Kernel Service

Purpose

Queries the existence of the Barrier Synchronization Register facility, and, if it exists, its size and allocation granule.

Syntax

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

int bsr_query (
 int *total_bytes,
 uint * supported_window_mask,
 int *free_bytes,
 uint *free_window_mask)
```

Parameters

Item	Description
total_bytes	Total bytes of the BSR facility currently present within the system or logical partition
supported_window_mask	Bit mask representing supported power-of-2-sized windows that can be allocated
free_bytes	Number of BSR bytes currently available (not allocated)
free_window_mask	Bit mask representing available (not allocated) power-of-2-sized windows

Description

The **bsr_query** service can be used to detect the presence and capabilities of the Barrier Synchronization Register (BSR) facility on a given system or logical partition. If the BSR facility is present on a system or within a logical partition, a value of `0` is returned, and the parameters, passed by reference, are written with the appropriate information.

The *total_bytes* field is written with the total number of BSR bytes currently present in the system or logical partition. The *supported_window_mask* field is written with a bitmask, where each bit set indicates the various power-of-2 window sizes that the *total_bytes* can be allocated and accessed. For example, a mask of `0x58` would indicate that windows of size 64 (0x40), 16 (0x10), and 8 (0x8) bytes were supported.

The *free_bytes* field is written with the number of BSR bytes within the system or logical partition that are currently unallocated. The *free_window_mask* field is written with a bitmask, where each bit set indicates the power-of-2 window sizes that are available for allocating and accessing the remaining *free_bytes*.

Note: Due to dynamic reconfiguration, the information returned by this query service might become stale.

Execution Environment

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

Return Values

Item	Description
0	The BSR facility exists and information is provided.
ENODEV	The BSR facility does not exist.

Related reference:

"bsr_alloc Kernel Service" on page 33

"bsr_free Kernel Service" on page 34

bwrite Kernel Service

Purpose

Writes the specified buffer data.

Syntax

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

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

Parameter

Item	Description
bp	Specifies the address of the buffer structure for the buffer to be written. On a platform that supports storage keys, the passed in *bp* parameter must be in the **KKEY_PUBLIC** or **KKEY_BLOCK_DEV** protection domain.

Description

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

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

Execution Environment

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

Return Values

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

Related reference:

"brelse Kernel Service" on page 32

"iowait Kernel Service" on page 232

Related information:

I/O Kernel Services

c

The following kernel services begin with the with the letter c.

cancel Device Queue Management Routine

Purpose

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

Syntax

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

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

Parameter

Item	Description
ptr	Specifies the address of the queue element.

Description

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

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

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

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

Execution Environment

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

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

Item	Description
cbp	Points to a **cfgncb** configuration notification control block. On a platform that supports storage keys, the passed in *cbp* parameter must only be in the **KKEY_PUBLIC** domain.

Description

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

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

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

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

Execution Environment

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

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

Related reference:

"cfgndel Kernel Service" on page 40

Related information:

sysconfig subroutine

Kernel Extension and Device Driver Management Kernel Services

cfgncb Configuration Notification Control Block

Purpose

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

Syntax

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

Parameters

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

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

Description

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

Each control block has the following definition:

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

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

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

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

If all registered notification routines create a return code of 0, then no value errors have been detected during validity checking. In this case, the **sysconfig** subroutine issues its second pass call to the **cfgncb.func** routine and sends the same parameters, although the *cmd* parameter contains a value of **CFGV_COMMIT**. This indicates that the new values go into effect at the earliest opportunity.

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

Related reference:

"cfgndel Kernel Service"

Related information:

SYS_SETPARMS subroutine

Kernel Extension and Device Driver Management Kernel Services

cfgndel Kernel Service

Purpose

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

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/sysconfig.h> void cfgndel (*cbp***) struct cfgncb ***cbp***;**

Parameter

Item	Description
cbp	Points to a **cfgncb** configuration notification control block. On a platform that supports storage keys, the passed in *cbp* parameter must only be in the **KKEY_PUBLIC** domain.

Description

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

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

Execution Environment

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

Return Values

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

Related reference:

"cfgnadd Kernel Service" on page 38

Related information:

sysconfig subroutine

Kernel Extension and Device Driver Management Kernel Services

check Device Queue Management Routine

Purpose

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

Syntax

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

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

Parameters

Item	Description
type	Specifies the type of call. The following values are used when the kernel calls the **check** routine: **CHECK_PARMS + SEND_CMD** Send command queue element. **CHECK_PARMS + START_IO** Start I/O CCB queue element. **CHECK_PARMS + GEN_PURPOSE** General purpose queue element.
ptr	Specifies the address of the queue element.
length	Specifies the length of the queue element.

Description

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

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

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

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

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

Execution Environment

The **check** routine executes under the process environment of the requester. Therefore, access to data areas must be handled as if the routine were in an interrupt handler environment. There is, however, no requirement to pin the code and data as in a normal interrupt handler environment.

Return Values

Item	Description
RC_GOOD	Indicates successful completion.

All other return values are device-specific.

Related reference:

"enque Kernel Service" on page 136

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

Item	Description
bp	Specifies the address of the buffer structure for the buffer to be cleared. On a platform that supports storage keys, the passed in *bp* parameter must be in the **KKEY_PUBLIC** or **KKEY_BLOCK_DEV** protection domain.

Description

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

Execution Environment

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

Return Values

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

Related information:

Block I/O Buffer Cache Kernel Services: Overview

I/O Kernel Services

clrjmpx Kernel Service

Purpose

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

Syntax

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

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

Parameter

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

Description

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

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

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

Execution Environment

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

Return Values

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

Related reference:

“setjmpx Kernel Service” on page 467

Related information:

Process and Exception Management Kernel Services

Understanding Exception Handling

common_reclock Kernel Service

Purpose

Implements a generic interface to the record locking functions.

Syntax

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

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

Parameters

Item — **Description**

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

size
Identifies the current size of the file in bytes.

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

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

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

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

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

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

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

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

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

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

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

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

Description

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

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

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

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

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

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

Execution Environment

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

Return Values

Item	Description
0	Indicates successful completion.
EAGAIN	Indicates a lock cannot be granted because of a blocking lock and the caller did not request that the operation sleep.
ERRNO	Indicates an error. Refer to the **fcntl** system call for the list of possible values.

Related information:

fcntl subroutine

flock.h subroutine

compare_and_swap Kernel Services

Purpose

Conditionally updates or returns a variable atomically.

Syntax

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

boolean_t compare_and_swap ( addr,  old_val_addr,  new_val)
atomic_p addr;
int * old_val_addr;
int new_val;

boolean_t compare_and_swaplp ( addr,  old_val_addr,  new_val)
atomic_l addr;
long * old_val_addr;
long new_val;
```

Parameters

Item	Description
addr	Specifies the address of the variable.
old_val_addr	Specifies the address of the old value to be checked against (and conditionally updated with) the value of the variable.
new_val	Specifies the new value to be conditionally assigned to the variable.

Description

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

The **compare_and_swap** kernel service operates on a single word (32 bit) variable while the **compare_and_swaplp** kernel service operates on a double word (64 bit) variable.

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

Note:

- The single word variable passed to the **compare_and_swap** kernel service must be aligned on a full word (32 bit) boundary.
- The double word variable passed to the **compare_and_swaplp** kernel service must be aligned on a double word (64 bit) boundary.

Execution Environment

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

Return Values

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

Related reference:

"fetch_and_add Kernel Services" on page 139

"fetch_and_and or fetch_and_or Kernel Services" on page 140

Related information:

Locking Kernel Services

coprocessor_user_register Kernel Service

Purpose

Registers the current process as a coprocessor user.

Syntax

```
#include <sys/coprocessor.h>
kerrno_t coprocessor_user_register ( int coprocessor_type, unsigned int * phandle )
```

Parameters

Item	Description
coprocessor_type	Numeric value in the [0..63] range
phandle	Pointer to an unsigned 32 bit integer where a handle identifying this process is returned.

Description

This kernel service allows a kernel extension to register the current process as a user of the coprocessor type passed as the first argument. When successful, the service sets up values in the process context that allow the current process to access coprocessors of the specified type in user mode.

Execution Environment

This kernel service can be called in the process environment only.

Return Values

When the call is successful, the kernel service returns a value of zero. Otherwise, a negative value is returned to indicate an error.

Error Values

Possible errors are:

- Coprocessors not supported (supported only on POWER7® and newer processors)
- Invalid coprocessor type (must be in the range 0-63).
- Bad address passed as the second argument.
- The current process is already registered for this coprocessor type.
- The service is being called in interrupt context.
- The service could not allocate a value for the handle.

Related information:

coprocessor_user_unregister subroutine

coprocessor_user_unregister Kernel Service

Purpose

Unregisters the current process as a coprocessor user.

Syntax

```
#include <sys/coprocessor.h>
kerrno_t coprocessor_user_unregister ( int coprocessor_type )
```

Parameters

Item	Description
coprocessor_type	Numeric value in the range [0..63] which identifies a coprocessor type.

Description

This kernel service allows a kernel extension to unregister the current process that was previously registered as a coprocessor user. When successful, further accesses by the process to the coprocessor type passed as an argument in user mode will fail with a privileged operation exception.

Execution Environment

This kernel service can be called in the process environment only.

Return Values

When the call is successful, the kernel service returns a value of zero. Otherwise, a negative value is returned to indicate an error.

Error Values

Possible errors are:

- Coprocessors not supported (supported only on POWER7 and newer processors).
- Invalid coprocessor type (must be in the range 0-63).
- The current process is not registered for this coprocessor type.
- The service is being called in interrupt context.

Related information:

coprocessor_user_register subroutine

copyin Kernel Service

Purpose

Copies data between user and kernel memory.

Syntax

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

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

Parameters

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

Description

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

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

Execution Environment

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

Return Values

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

Related reference:

"copyinstr Kernel Service"

"copyout Kernel Service" on page 50

Related information:

Accessing User-Mode Data While in Kernel Mode

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

On the 32-bit kernel, the syntax for the **copyinstr** Kernel Service is:

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

On the 64-bit kernel, the syntax for the **copyinstr** subroutine is:

```
int copyinstr (from, to, max, actual)
void *from;
void *to;
size_t max;
size_t *actual;
```

Parameters

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

Description

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

Execution Environment

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

Return Values

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

Related information:

Accessing User-Mode Data While in Kernel Mode

Memory Kernel Services

copyout Kernel Service

Purpose

Copies data between user and kernel memory.

Syntax

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

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

Parameters

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

Description

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

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

Execution Environment

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

Return Values

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

Related reference:
"copyin Kernel Service" on page 48
"copyinstr Kernel Service" on page 49
Related information:
Memory Kernel Services

crcopy Kernel Service

Purpose

Copies a credentials structure to a new one and frees the old one.

Syntax

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

```
struct ucred * crcopy ( cr)
struct ucred * cr;
```

Parameter

Item	Description
cr	Pointer to the credentials structure that is to be copied and then freed.

Description

The **crcopy** kernel service allocates a new credentials structure that is initialized from the contents of the *cr* parameter. The reference to *cr* is then freed and a pointer to the new structure returned to the caller.

Note: The *cr* parameter must have been obtained by an earlier call to the **crcopy** kernel service, **crdup** kernel service, **crget** kernel service, or the **crref** kernel service.

Execution Environment

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

Return Values

Item	Description
Nonzero value	A pointer to a newly allocated and initialized credentials structure.
Zero value	An error occurred when the kernel service was attempting to allocate pinned memory for the credentials structure.

Related information:

Security Kernel Services

crdup Kernel Service

Purpose

Copies a credentials structure to a new one.

Syntax

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

```
struct ucred * crdup ( cr)
struct ucred * cr;
```

Parameter

Item	Description
cr	Pointer to the credentials structure that is to be copied.

Description

The **crdup** kernel service allocates a new credentials structure that is initialized from the contents of the *cr* parameter.

Execution Environment

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

Return Values

Item	Description
Nonzero value	A pointer to a newly allocated and initialized credentials structure.
Zero value	An error occurred when the kernel service was attempting to allocate pinned memory for the credentials structure.

Related information:

Security Kernel Services

creatp Kernel Service

Purpose

Creates a new kernel process.

Syntax

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

pid_t creatp()
```

Description

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

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

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

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

Execution Environment

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

Return Values

Item	Description
-1	Indicates an error.

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

Related reference:

“initp Kernel Service” on page 220

Related information:

Process and Exception Management Kernel Services

Using Kernel Processes

CRED_GETEUID, CRED_GETRUID, CRED_GETSUID, CRED_GETLUID, CRED_GETEGID, CRED_GETRGID, CRED_GETSGID and CRED_GETNGRPS Macros

Purpose

Credentials structure field accessing macros.

Syntax

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

uid_t CRED_GETEUID ( crp )
uid_t CRED_GETRUID ( crp )
uid_t CRED_GETSUID ( crp )
uid_t CRED_GETLUID ( crp )
gid_t CRED_GETEGID ( crp )
gid_t CRED_GETRGID ( crp )
gid_t CRED_GETSGID ( crp )
int CRED_GETNGRPS ( crp )
```

Parameter

Item	Description
crp	Pointer to a credentials structure

Description

These macros provide a means for accessing the user and group identifier fields within a credentials structure. The fields within a **ucred** structure should not be accessed directly as the field names and their locations are subject to change.

The **CRED_GETEUID** macro returns the effective user ID field from the credentials structure referenced by *crp*.

The **CRED_GETRUID** macro returns the real user ID field from the credentials structure referenced by *crp*.

The **CRED_GETSUID** macro returns the saved user ID field from the credentials structure referenced by *crp*.

The **CRED_GETLUID** macro returns the login user ID field from the credentials structure referenced by *crp*.

The **CRED_GETEUID** macro returns the effective group ID field from the credentials structure referenced by *crp*.

The **CRED_GETRUID** macro returns the real group ID field from the credentials structure referenced by *crp*.

The **CRED_GETSUID** macro returns the saved group ID field from the credentials structure referenced by *crp*.

The **CRED_GETNGRPS** macro returns the number of concurrent group ID values stored within the credentials structure referenced by *crp*.

These macros are defined in the system header file <*sys/cred.h*>.

Execution Environment

The credentials macros called with any valid credentials pointer.

Related information:

Security Kernel Services

crexport Kernel Service

Purpose

Copies an internal format credentials structure to an external format credentials structure.

Syntax

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

void crexport (src, dst)
struct ucred * src;
struct ucred_ext * dst;
```

Parameter

Item	Description
src	Pointer to the internal credentials structure.
dst	Pointer to the external credentials structure.

Description

The **crexport** kernel service copies from the internal credentials structure referenced by *src* into the external credentials structure referenced by *dst*. The external credentials structure is guaranteed to be compatible between releases. Fields within a **ucred** structure must not be referenced directly as the field names and locations within that structure are subject to change.

Execution Environment

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

Return Values

This kernel service does not have a return value.

Related information:

Security Kernel Services

crfree Kernel Service

Purpose

Releases a reference count on a credentials structure.

Syntax

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

void crfree ( cr)
struct ucred * cr;
```

Parameter

Item	Description
cr	Pointer to the credentials structure that is to have a reference freed.

Description

The **crfree** kernel service deallocates a reference to a credentials structure. The credentials structure is deallocated when no references remain.

Note: The *cr* parameter must have been obtained by an earlier call to the **crcopy** kernel service, **crdup** kernel service, **crget** kernel service, or the **crref** kernel service.

Execution Environment

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

Return Values

No value is returned by this kernel service.

Related information:

Security Kernel Services

crget Kernel Service

Purpose

Allocates a new, uninitialized credentials structure to a new one and frees the old one.

Syntax

```
#include <sys/cred.h>
struct ucred * crget ( void )
```

Parameter

This kernel service does not require any parameters.

Description

The **crget** kernel service allocates a new credentials structure. The structure is initialized to all zero values, and the reference count is set to 1.

Execution Environment

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

Return Values

Item	Description
Nonzero value	A pointer to a newly allocated and initialized credentials structure.
Zero value	An error occurred when the kernel service was attempting to allocate pinned memory for the credentials structure.

Related information:

Security Kernel Services

crhold Kernel Service

Purpose

Increments the reference count for a credentials structure.

Syntax

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

```
void crhold ( cr)
struct ucred * cr;
```

Parameter

Item	Description
cr	Pointer to the credentials structure that will have its reference count incremented.

Description

The **crhold** kernel service increments the reference count of a credentials structure.

Note: Reference counts that are incremented with the **crhold** kernel service must be decremented with the **crfree** kernel service.

Execution Environment

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

Return Values

No value is returned by this kernel service.

Related information:

Security Kernel Services

crref Kernel Service

Purpose

Increments the reference count for the current credentials structure.

Syntax

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

```
struct ucred * crref ( void )
```

Parameter

This kernel service does not require any parameters.

Description

The **crref** kernel service increments the reference count of the current credentials structure and returns a pointer to the current credentials structure to the invoker.

Note: References that are allocated with the **crref** kernel service must be released with the **crfree** kernel service.

Execution Environment

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

Return Values

Item	Description
Nonzero value	A pointer to the current credentials structure.

Related information:
Security Kernel Services

crset Kernel Service

Purpose

Sets the current security credentials.

Syntax

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

void crset ( cr)
struct ucred * cr;
```

Parameter

Item	Description
cr	Pointer to the credentials structure that will become the new, current security credentials.

Description

The **crset** kernel service replaces the current security credentials with the supplied value. The existing structure will be deallocated.

Note: The *cr* parameter must have been obtained by an earlier call to the **crcopy** kernel service, **crdup** kernel service, **crget** kernel service, or the **crref** kernel service.

Execution Environment

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

Return Values

No value is returned by this kernel service.

Related information:

Security Kernel Services

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

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

Description

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

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

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

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

Execution Environment

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

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

Return Values

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

Related information:

Timer and Time-of-Day Kernel Services

d

The following kernel services begin with the with the letter d.

d_align Kernel Service

Purpose

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

Library

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

Syntax

```
int  d_align()
```

Description

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

Execution Environment

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

Related reference:

“d_cflush Kernel Service” on page 61

“d_roundup Kernel Service” on page 103

Related information:

Understanding Direct Memory Access (DMA) Transfer

d_alloc_dmamem Kernel Service

Purpose

Allocates an area of “dma-able” memory.

Syntax

```
void *  d_alloc_dmamem(d_handle_t  device_handle, size_t size,int align)
```

Description

Exported, documented kernel service supported on PCI-based systems only. The **d_alloc_dmamem** kernel service allocates an area of “dma-able” memory which satisfies the constraints associated with a DMA handle, specified via the *device_handle* parameter. The constraints (such as need for contiguous physical pages or need for 32-bit physical address) are intended to guarantee that a given adapter will be able to access the physical pages associated with the allocated memory. A driver associates such constraints with a dma handle via the *flags* parameter on its **d_map_init** call.

The area to be allocated 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 12 requests that the allocated area be aligned on a 4096 byte boundary.

d_alloc_dmamem is appropriate to be used for long-term mappings. Depending on the system configuration and the constraints encoded in the *device_handle*, the underlying storage will come from either the real_heap (**rmalloc** service) or pinned_heap (**xmalloc** service).

Note:

1. The **d_free_dmamem** service should be called to free allocation from a previous **d_alloc_dmamem** call.
2. The **d_alloc_dmamem** kernel service can be called from the process environment only.

Parameters

Item	Description
device_handle	Indicates the dma handle.
align	Specifies alignment characteristics.
size_t size	Specifies number of bytes to allocate.

Return Values

Item	Description
Address of allocated area	Indicates that **d_alloc_dmamem** was successful.
NULL	Requested memory could not be allocated.

Related reference:

"d_free_dmamem Kernel Service" on page 75

"d_map_init Kernel Service" on page 81

"rmalloc Kernel Service" on page 448

d_cflush Kernel Service

Purpose

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

Syntax

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

Parameters

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

Description

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

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

Long-Term DMA Buffer Mapping

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

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

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

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

Note: The **d_cflush** kernel service is not supported on the 64-bit kernel.

Execution Environment

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

Return Values

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

Related information:

I/O Kernel Services

Understanding Direct Memory Access (DMA) Transfer

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

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

Description

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

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

Execution Environment

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

Return Values

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

Related information:

Timer and Time-of-Day Kernel Services

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

Item	Description
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

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

Example

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

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

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

Related reference:

"add_domain_af Kernel Service" on page 8

Related information:

Network Kernel Services

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

Item	Description
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

Item	Description
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.

Related reference:

"add_input_type Kernel Service" on page 8

"find_input_type Kernel Service" on page 142

Related information:

Network Kernel Services

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

Item	Description
soft_intr_level	Specifies the software interrupt level to delete. This parameter must be greater than or equal to 0 and less than **NETISR_MAX**. Refer to **netisr.h** for the range of values of *soft_intr_level* that are already in use. Also, other kernel extensions that are not AIX and that use network ISRs currently running on the system can make use of additional values not mentioned in **netisr.h**.

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

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

Example

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

```
del_netisr(NETISR_IP);
```

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

Related reference:

"add_netisr Kernel Service" on page 10

Related information:

Network Kernel Services

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

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

Description

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

Execution Environment

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

Return Values

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

Related reference:

"add_netopt Macro" on page 10

Related information:

no command

Network Kernel Services

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

Item	Description
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

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

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

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

Execution Environment

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

Return Values

Item	Description
RC_GOOD	Indicates successful completion.

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

devdump Kernel Service

Purpose

Calls a device driver dump-to-device routine.

Syntax

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

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

Parameters

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

Description

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

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

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

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

Execution Environment

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

Return Values

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

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

Related reference:

"dddump Device Driver Entry Point" on page 623

Related information:

Kernel Extension and Device Driver Management Kernel Services

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

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

Description

The kernel or kernel extension calls the **devstrat** kernel service to request a block data transfer to or from the device with the specified device number. This device number is found in the **buf** structure. The **devstrat** service can only be used for the block class of device drivers.

The **devstrat** service calls the device driver's **ddstrategy** routine. This routine is found in the device switch table for the device driver associated with the specified device number found in the `b_dev` field. The `b_dev` field is found in the **buf** structure pointed to by the *bp* parameter. The caller of the **devstrat** service must have an **iodone** routine specified in the `b_iodone` field of the **buf** structure. Following the return from the device driver's **ddstrategy** routine, the **devstrat** service returns without waiting for the I/O to be performed.

On multiprocessor systems, all **iodone** routines run by default on the first processor started when the system was booted. This ensures compatibility with uniprocessor device drivers. If the **iodone** routine has been designed to be multiprocessor-safe, set the **B_MPSAFE** flag in the `b_flags` field of the **buf** structure passed to the **devstrat** kernel service. The **iodone** routine will then run on any available processor.

If the device major number is not valid or the specified device is not a block device driver, the **devstrat** service returns the **ENODEV** return code. If the device number is valid, the device driver's **ddstrategy** routine is called with the pointer to the **buf** structure (specified by the *bp* parameter).

Execution Environment

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

Note: The **devstrat** kernel service can be called in the interrupt environment only if its priority level is **INTIODONE** or lower.

Return Values

Item	Description
0	Indicates a successful operation.
ENODEV	Indicates that the device number is not valid or that no **ddstrategy** routine registered. This value is also returned when the specified device is not a block device driver. If this error occurs, the **devstrat** service can cause a page fault.

Related reference:

“ddstrategy Device Driver Entry Point” on page 634

“buf Structure” on page 614

Related information:

Kernel Extension and Device Driver Management Kernel Services

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

Item	Description
devno	Specifies the major and minor device numbers to be associated with the specified entry in the device switch table.
dswptr	Points to the device switch structure to be added to the device switch table.

Description

The **devswadd** kernel service is typically called by a device driver's **ddconfig** routine to add or replace the device driver's entry points in the device switch table. The device switch table is a table of device switch (**devsw**) structures indexed by the device driver's major device number. This table of structures is used by the device driver interface services in the kernel to facilitate calling device driver routines.

The major device number portion of the *devno* parameter is used to specify the index in the device switch table where the **devswadd** service must place the specified device switch entry. Before this service copies the device switch structure into the device switch table, it checks the existing entry to determine if any opened device is using it. If an opened device is currently occupying the entry to be replaced, the **devswadd** service does not perform the update. Instead, it returns an **EEXIST** error value. If the update is successful, it returns a value of 0.

Entry points in the device switch structure that are not supported by the device driver must be handled in one of two ways. If a call to an unsupported entry point should result in the return of an error code, then the entry point must be set to the **nodev** routine in the structure. As a result, any call to this entry point automatically invokes the **nodev** routine, which returns an **ENODEV** error code. The kernel provides the **nodev** routine.

Otherwise, a call to an unsupported entry point should be treated as a no-operation function. Then the corresponding entry point should be set to the **nulldev** routine. The **nulldev** routine, which is also provided by the kernel, performs no operation if called and returns a 0 return code.

On multiprocessor systems, all device driver routines run by default on the first processor started when the system was booted. This ensures compatibility with uniprocessor device drivers. If the device driver being added has been designed to be multiprocessor-safe, set the **DEV_MPSAFE** flag in the `d_opts` field of the **devsw** structure passed to the **devswadd** kernel service. The device driver routines will then run on any available processor.

All other fields within the structure that are not used should be set to 0. Some fields in the structure are for kernel use; the **devswadd** service does not copy these fields into the device switch table. These fields are documented in the **/usr/include/device.h** file.

Execution Environment

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

Return Values

Item	Description
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.

Related reference:

"devswdel Kernel Service" on page 72

"ddconfig Device Driver Entry Point" on page 620

Related information:

Kernel Extension and Device Driver Management Kernel Services

devswchg Kernel Service

Purpose

Alters a device switch entry point in the device switch table.

Syntax

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

int devswchg ( devno,  type,  newfunc,  oldfunc);
dev_t devno;
int type;
int (*newfunc) ();
int (**oldfunc)();
```

Parameters

Item	Description
devno	Specifies the major and minor device numbers of the device to be changed.
type	Specifies the device switch entry point to alter. The *type* parameter can have one of the following values: **DSW_BLOCK** Alters the **ddstrategy** entry point. **DSW_CONFIG** Alters the **ddconfig** entry point. **DSW_CREAD** Alters the **ddread** entry point. **DSW_CWRITE** Alters the **ddwrite** entry point. **DSW_DUMP** Alters the **dddump** entry point. **DSW_MPX** Alters the **ddmpx** entry point. **DSW_SELECT** Alters the **ddselect** entry point. **DSW_TCPATH** Alters the **ddrevoke** entry point.
newfunc	Specifies the new value for the device switch entry point.
oldfunc	Specifies that the old value of the device switch entry point be returned here.

Description

The **devswchg** kernel service alters the value of a device switch entry point (function pointer) after a device switch table entry has been added by the **devswadd** kernel service. The device switch entry point specified by the *type* parameter is set to the value of the *newfunc* parameter. Its previous value is returned in the memory addressed by the *oldfunc* parameter. Only one device switch entry can be altered per call.

If the **devswchg** kernel service is unsuccessful, the value referenced by the *oldfunc* parameter is not defined.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates the *Type* command was not valid.
ENODEV	Indicates the device switch entry specified by the *devno* parameter is not defined.

Related reference:

"devswadd Kernel Service" on page 70

Related information:

List of Kernel Extension and Device Driver Management Kernel Services

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

Item	Description
devno	Specifies the major and minor device numbers of the device to be deleted.

Description

The **devswdel** kernel service is typically called by a device driver's **ddconfig** routine on termination to remove the device driver's entry points from the device switch table.The device switch table is a table of device switch (**devsw**) structures indexed by the device driver's major device number. The device driver interface services use this table of structures in the kernel to facilitate calling device driver routines.

The major device number portion of the *devno* parameter is used to specify the index into the device switch table for the entry to be removed. Before the device switch structure is removed, the existing entry is checked to determine if any opened device is using it.

If an opened device is currently occupying the entry to be removed, the **devswdel** service does not perform the update. Instead, it returns an **EEXIST** return code. If the removal is successful, a return code of 0 is set.

The **devswdel** service removes a device switch structure entry from the table by marking the entry as undefined and setting all of the entry point fields within the structure to a **nodev** value. As a result, any callers of the removed device driver return an **ENODEV** error code. If the specified entry is already marked undefined, the **devswdel** service returns an **ENODEV** error code.

Execution Environment

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

Return Values

Item	Description
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.

Related reference:

"devswchg Kernel Service" on page 71

"devswqry Kernel Service" on page 74

Related information:

Kernel Extension and Device Driver Management Kernel Services

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

Item	Description
devno	Specifies the major and minor device numbers of the device to be queried.
status	Points to the status of the specified device entry in the device switch table. This parameter is passed by reference.
dsdptr	Points to device-dependent information for the specified device entry in the device switch table. This parameter is passed by reference.

Description

The **devswqry** kernel service returns the status of a specified device entry in the device switch table. The entry in the table to query is determined by the major portion of the device number specified in the *devno* parameter. The status of the entry is returned in the *status* parameter that is passed by reference on the call. If this pointer is null on entry to the **devswqry** service, then the status is not returned to the caller.

The **devswqry** service also returns the address of device-dependent information for the specified device entry in the device switch table. This address is taken from the `d_dsdptr` field for the entry and returned in the *dsdptr* parameter, which is passed by reference. If this pointer is null on entry to the **devswqry** service, then the service does not return the address from the `d_dsdptr` field to the caller.

Status Parameter Flags

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

Item	Description
DSW_BLOCK	Device switch entry is defined by a block device driver. This flag is set when the device driver has a **ddstrategy** entry point.
DSW_CONFIG	Device driver in this device switch entry provides an entry point for configuration.
DSW_CREAD	Device driver in this device switch entry is providing a routine for character reads or raw input. This flag is set when the device driver has a **ddread** entry point.
DSW_CWRITE	Device driver in this device switch entry is providing a routine for character writes or raw output. This flag is set when the device driver has a **ddwrite** entry point.
DSW_DEFINED	Device switch entry is defined.
DSW_DUMP	Device driver defined by this device switch entry provides the capability to support one or more of its devices as targets for a kernel dump. This flag is set when the device driver has provided a **dddump** entry point.
DSW_MPX	Device switch entry is defined by a multiplexed device driver. This flag is set when the device driver has a **ddmpx** entry point.
DSW_OPENED	Device switch entry is in use and the device has outstanding opens. This flag is set when the device driver has at least one outstanding open.
DSW_SELECT	Device driver in this device switch entry provides a routine for handling the **select** or **poll** subroutines. This flag is set when the device driver has provided a **ddselect** entry point.

Item	Description
DSW_TCPATH	Device driver in this device switch entry supports devices that are considered to be in the trusted computing path and provide support for the revoke function. This flag is set when the device driver has provided a **ddrevoke** entry point.
DSW_TTY	Device switch entry is in use by a tty device driver. This flag is set when the pointer to the **d_ttys** structure is not a null character**.**
DSW_UNDEFINED	Device switch entry is not defined.

The *status* parameter is set to the **DSW_UNDEFINED** flag when a device switch entry is not in use. This is the case if either of the following are true:

- The entry has never been used. (No previous call to the **devswadd** service was made.)
- The entry has been used but was later deleted. (A call to the **devswadd** service was issued, followed by a call to the **devswdel** service.)

No other flags are set when the **DSW_UNDEFINED** flag is set.

Note: The *status* parameter must be a null character if called from the interrupt environment.

Execution Environment

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

Return Values

Item	Description
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.

Related reference:

"devswadd Kernel Service" on page 70

"devswchg Kernel Service" on page 71

Related information:

Kernel Extension and Device Driver Management Kernel Services

d_free_dmamem Kernel Service

Purpose

Frees an area of memory.

Syntax

```
int d_free_dmamem(d_handle_t  device_handle, void * addr, size_t size)
```

Description

Exported, documented kernel service supported on PCI-based systems only. The **d_free_dmamem** kernel service frees the area of memory pointed to by the *addr* parameter. This area of memory must be allocated with the **d_alloc_dmamem** kernel service using the same *device_handle*, and the *addr* must be the address returned from the corresponding **d_alloc_dmamem** call. Also, the size must be the same size that was used on the corresponding **d_alloc_dmamem** call.

Note:

1. Any memory allocated in a prior **d_alloc_dmamem** call must be explicitly freed with a **d_free_dmamem** call.

2. This service can be called from the process environment only.

Parameters

Item	Description
device_handle	Indicates the dma handle.
size_t size	Specifies size of area to free.
*void * addr*	Specifies address of area to free.

Return Values

Item	Description
0	Indicates successful completion.
–1	Indicates underlying free service (xmfree or rmalloc) failed.

Related reference:

"d_alloc_dmamem Kernel Service" on page 60

disable_lock Kernel Service

Purpose

Raises the interrupt priority, and locks a simple lock if necessary.

Syntax

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

int disable_lock ( int_pri,  lock_addr)
int int_pri;
simple_lock_t lock_addr;
```

Parameters

Item	Description
int_pri	Specifies the interrupt priority to set.
lock_addr	Specifies the address of the lock word to lock.

Description

The **disable_lock** kernel service raises the interrupt priority, and locks a simple lock if necessary, in order to provide optimized thread-interrupt critical section protection for the system on which it is executing. On a multiprocessor system, calling the **disable_lock** kernel service is equivalent to calling the **i_disable** and **simple_lock** kernel services. On a uniprocessor system, the call to the **simple_lock** service is not necessary, and is omitted. However, you should still pass a valid lock address to the **disable_lock** kernel service. Never pass a **NULL** lock address.

Execution Environment

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

Return Values

The **disable_lock** kernel service returns the previous interrupt priority.

Related reference:

"i_disable Kernel Service" on page 207

"simple_lock_init Kernel Service" on page 474

Related information:

Understanding Locking

disablement_checking_resume Kernel Service

Purpose

Indicates the end of a disabled code path that was exempted from detection of excessive interrupt disablement.

Syntax

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

void disablement_checking_resume(long prev_state)
```

Parameters

Item	Description
prev_state	Specifies the disablement detection state to be restored. This value is returned by the **disablement_checking_suspend** kernel service.

Description

The **disablement_checking_resume** service restores the disablement detection state to the value passed as *prev_state*. This service must be called after reenabling interrupts at the end of an INTMAX critical section, not within it. This is because, in the case of an INTMAX critical section, the tick counting will have been deferred by the total disablement until the moment of enablement.

This service must be used in conjunction with the **disablement_checking_suspend** kernel service, which temporarily stops disablement detection.

Note: Error checking, including that for excessive interrupt disablement, can be enabled or disabled by the **errctrl** command.

Execution Environment

The **disablement_checking_resume** service can be called from either the process or the interrupt environments.

Related reference:

"disablement_checking_suspend Kernel Service"

Related information:

errctrl command

disablement_checking_suspend Kernel Service

Purpose

Indicates the start of a disabled code path that is exempt from detection of excessive interrupt disablement.

Syntax

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

long disablement_checking_suspend(void)
```

Description

A call to the **disablement_checking_suspend** service temporarily disables the detection of excessive disablement for the duration of a portion of a critical section. For base level code, insert this call at the beginning of the exempt critical section immediately after it disables, or as soon as possible within interrupt handling code.

This service must be used in conjunction with the **disablement_checking_resume** kernel service, which resumes the prior disablement checking state.

Note: Error checking, including that for excessive interrupt disablement, can be enabled or disabled by the **errctrl** command.

Execution Environment

The **disablement_checking_suspend** service can be called from either the process or the interrupt environments. Interrupts should be at least partially disabled at the time of the call.

Return Values

The **disablement_checking_suspend** service returns the previous suspension state to the caller. This value must be passed later to the resume function, which restores that state. This enables nesting of exempt critical sections.

Related reference:

"disablement_checking_resume Kernel Service" on page 77

Related information:

errctrl command

d_map_attr Kernel Service

Purpose

Changes the attributes associated with a DMA handle.

Syntax

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

kerrno_t d_map_attr (handle, cmd, attr, attr_size)
d_handle_t handle;
ulong cmd;
void * attr;
size_t attr_size;
```

Parameters

Item	Description
handle	Indicates the unique handle returned by the **d_map_init_ext** kernel service.
cmd	Specifies one of the following flags: **D_ATTR_SET_MIN_MAPMEM** Sets the minimum amount of I/O mappable memory. This is the logical memory change and not the DMA bus memory change. **D_ATTR_SET_DES_MAPMEM** Sets the desired amount of I/O mappable memory. This is the logical memory change and not the DMA bus memory change.
attr	You must set this parameter to the value of **size64_t ***. This parameter sets the minimum or the desired amount of I/O mappable memory depending on the specified value of the *cmd* parameter.
attr_size	You must set this parameter to the value of **sizeof(size64_t)**. This parameter sets the minimum or the desired amount of I/O mappable memory depending on the specified value of the *cmd* parameter.

Description

The **d_map_attr** kernel service can change certain attributes of the **d_handle_t** structure in case the needs of a device driver change during runtime. For example, if a device driver needs more DMA space at runtime, it can call the **d_map_attr** kernel service to request an increase in the map space. The **d_map_attr** kernel service is not an exported kernel service, but a bus specific utility routine determined by the **d_map_init_ext** kernel service and provided to the caller through the **d_handle** structure.

Execution Environment

The **d_map_attr** kernel service can be called from the process environment at **INTBASE**. Serialization with other DMA services like the **d_map_page** service and the **d_unmap_page** service is the caller's responsibility.

Return Values

Item	Description
DMA_SUCC	Indicates a successful completion.
EINVAL_D_MAP_ATTR	Indicates that the specified *cmd* parameter is not valid.
ENOMEM_D_MAP_ATTR	Indicates that it is unable to change the minimum or desired I/O mappable memory.

Related reference:

"d_map_init_ext Kernel Service" on page 82

d_map_clear Kernel Service

Purpose

Deallocates resources previously allocated on a **d_map_init** call.

Syntax

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

void d_map_clear (*handle)
struct d_handle *handle
```

Parameters

Item	Description
handle	Indicates the unique handle returned by the **d_map_init** kernel service.

Description

The **d_map_clear** kernel service is a bus-specific utility routine determined by the **d_map_init** service that deallocates resources previously allocated on a **d_map_init** call. This includes freeing the **d_handle** structure that was allocated by **d_map_init**.

Note: You can use the **D_MAP_CLEAR** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_map_clear** kernel service.

Related reference:

"d_map_init Kernel Service" on page 81

d_map_disable Kernel Service

Purpose

Disables DMA for the specified handle.

Syntax

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

int d_map_disable(*handle)
struct d_handle *handle;
```

Parameters

Item	Description
handle	Indicates the unique handle returned by **d_map_init**.

Description

The **d_map_disable** kernel service is a bus-specific utility routine determined by the **d_map_init** kernel service that disables DMA for the specified *handle* with respect to the platform.

Note: You can use the **D_MAP_DISABLE** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_map_disable** kernel service.

Return Values

Item	Description
DMA_SUCC	Indicates the DMA is successfully disabled.
DMA_FAIL	Indicates the DMA could not be explicitly disabled for this device or bus.

Related reference:

"d_map_init Kernel Service" on page 81

d_map_enable Kernel Service

Purpose

Enables DMA for the specified handle.

Syntax

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

int d_map_enable(*handle)
struct d_handle *handle;
```

Parameters

Item	Description
handle	Indicates the unique handle returned by **d_map_init**.

Description

The **d_map_enable** kernel service is a bus-specific utility routine determined by the **d_map_init** kernel service that enables DMA for the specified *handle* with respect to the platform.

Note: You can use the **D_MAP_ENABLE** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_map_enable** kernel service.

Return Values

Item	Description
DMA_SUCC	Indicates the DMA is successfully enabled.
DMA_FAIL	Indicates the DMA could not be explicitly enabled for this device or bus.

Related reference:

"d_map_init Kernel Service"

d_map_init Kernel Service

Purpose

Allocates and initializes resources for performing DMA with PCI and ISA devices.

Syntax

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

struct d_handle* d_map_init (bid, flags, bus_flags, channel)
int bid;
int flags;
int bus_flags;
uint channel;
```

Parameters

Item	Description
bid	Specifies the bus identifier.
flags	Describes the mapping.
bus_flags	Specifies the target bus flags.
channel	Indicates the *channel* assignment specific to the bus.

Description

The **d_map_init** kernel service allocates and initializes resources needed for managing DMA operations and returns a unique *handle* to be used on subsequent DMA service calls. The *handle* is a pointer to a **d_handle** structure allocated by **d_map_init** from the pinned heap for the device. The device driver uses the function addresses provided in the *handle* for accessing the DMA services specific to its host bus. The **d_map_init** service returns a **DMA_FAIL** error when resources are unavailable or cannot be allocated.

The *channel* parameter is the assigned channel number for the device, if any. Some devices and or buses might not have the concept of *channels*. For example, an ISA device driver would pass in its assigned DMA channel in the *channel* parameter.

Note: The possible flag values for the *flags* parameter can be found in **/usr/include/sys/dma.h**. These flags can be logically ORed together to reflect the desired characteristics.

Execution Environment

The **d_map_init** kernel service should only be called from the process environment.

Return Values

Item	Description
DMA_FAIL	Indicates that the resources are unavailable. No registration was completed.
struct d_handle *	Indicates successful completion.

Related reference:

"d_unmap_page Kernel Service" on page 107

"d_map_list Kernel Service" on page 83

"d_map_disable Kernel Service" on page 79

d_map_init_ext Kernel Service

Purpose

Allocates and initializes resources for performing DMA with PCI and VDEVICE devices.

Syntax

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

kerrno_t d_map_init_ext (dma_input, info_size, handle_ptr)
d_info_t * dma_input;
size_t info_size;
d_handle_t * handle_ptr;
```

Parameters

Item	Description
dma_input	Contains information like the bus identifier, flags, and so on.
info_size	Specifies the size of the *dma_input* parameter in bytes.
handle_ptr	Contains the DMA handle returned upon success.

Description

The **d_map_init_ext** kernel service is very similar to the **d_map_init** kernel service. Unlike the **d_map_init** kernel service, the input argument list of the **d_map_init_ext** kernel service is not limited and can be extended without breaking binary compatibility. Also, the **d_map_init_ext** kernel service returns a **kerrno_t** type return code which contains more RAS information rather than just the **DMA_FAIL** value.

The caller of the **d_map_init_ext** kernel service initializes the **d_info_t** structure and passes it into the **d_map_init_ext** kernel service by reference. The size of the **d_info_t** type must match the *info_size* parameter. This allows future expansion of the **d_info_t** type safely. If there is a size mismatch, the **d_map_init_ext** kernel service fails. The **d_map_init_ext** kernel service also creates a new private pool of I/O memory entitlement that can be used for DMA. The private pool is created by carving out a chunk of total I/O memory entitlement for the AIX partition. Thus, in order to create a **d_handle_t** type successfully, there must be sufficient DMA PCI space and I/O memory entitlement.

The following structure is defined in the **sys/dma.h** file:

```
#define DMA_MAX_MAPPER_NAME     32
typedef struct d_info
{
      uint64_t    di_bid;
      uint64_t    di_flags;
      uint64_t    di_bus_flags;
      uint64_t    di_channel;
      uint64_t    di_min_mapmem;
      uint64_t    di_des_mapmem;
      uint64_t    di_max_mapmem;
      char        di_mapper_name[DMA_MAX_MAPPER_NAME];
} d_info_t;
```

Note: The first four fields of the **d_info_t** type match the four arguments of the **d_map_init** kernel service. Therefore, all flags and bus_flags on the **d_map_init** kernel service are honored by the **d_map_init_ext** kernel service except the *DMA_MAXMIN_** flags. The *DMA_MAXMIN_** flags are replaced with the *di_min_mapmem*, *di_des_mapmem*, and *di_max_mapmem* fields. They not only specify the required amount of DMA space, but also the necessary I/O memory entitlement for the device.

The *di_min_mapmem* parameter is the minimum amount of memory that the driver must be able to map for DMA in order to ensure the forward progress. The **d_map_init_ext** kernel service fails if the minimum I/O memory entitlement requirement cannot be satisfied.

The *di_des_mapmem* parameter is the required amount of memory that the driver wants to be able to I/O map in order to have good throughput. In most cases, this is a value that a driver specifies through the *DMA_MAXMIN_** flag.

The *di_max_mapmem* parameter is the maximum amount of memory that the driver can ever map for DMA. This is the amount of DMA space that the **d_map_init_ext** kernel service can allocate.

Note: While the I/O memory entitlement for a **d_handle_t** type can be changed at runtime through the **d_map_attr** kernel service, the DMA space cannot be changed dynamically.

The *di_mapper_name* parameter contains the name of the device instance using the DMA resources (for example, ent0, scsi1, and so on).

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
struct d_handle *	Indicates a successful completion.
ENOMEM_D_MAP_INIT_EXT_1	Indicates that the memory allocation failed. An AIX error is logged.
ENOMEM_D_MAP_INIT_EXT_2	Cannot reserve I/O memory entitlement with the least amount specified by the *di_min_mapmem* parameter. An AIX error is logged.
ENOMEM_D_MAP_INIT_EXT_3	Cannot allocate enough DMA space. An AIX error is logged.
EINVAL_D_MAP_INIT_EXT_1	Indicates that some input argument is not valid. An AIX error is logged in some cases.
EINVAL_D_MAP_INIT_EXT_2	Indicates that the combination of input arguments and system configuration is not valid. No AIX error is logged.
EINVAL_D_MAP_INIT_EXT_3	Indicates that the RAS initialization failed. No AIX error is logged.

Related reference:

"d_map_clear Kernel Service" on page 79

"d_unmap_page Kernel Service" on page 107

"d_map_list Kernel Service"

d_map_list Kernel Service

Purpose

Performs platform-specific DMA mapping for a list of virtual addresses.

Syntax

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

int d_map_list (*handle, flags, minxfer, *virt_list, *bus_list)
struct d_handle *handle;
int flags;
int minxfer;
struct dio *virt_list;
struct dio *bus_list;
```

Note: The following is the interface definition for **d_map_list** when the **DMA_ADDRESS_64** and **DMA_ENABLE_64** flags are set on the **d_map_init** call.

```
int d_map_list (*handle, flags, minxfer, *virt_list, *bus_list)
struct d_handle *handle;
int flags;
int minxfer;
struct dio_64 *virt_list;
struct dio_64 *bus_list;
```

Parameters

Item	Description
handle	Indicates the unique handle returned by the **d_map_init** kernel service.
flags	Specifies one of the following flags: **DMA_READ** Transfers from a device to memory. **BUS_DMA** Transfers from one device to another device. **DMA_BYPASS** Do not check page access. **DMA_STMAP** Indicates a short-term mapping.
minxfer	Specifies the minimum transfer size for the device.
virt_list	Specifies a list of virtual buffer addresses and lengths.
bus_list	Specifies a list of bus addresses and lengths.

Description

The **d_map_list** kernel service is a bus-specific utility routine determined by the **d_map_init** kernel service that accepts a list of virtual addresses and sizes and provides the resulting list of bus addresses. This service fills out the corresponding bus address list for use by the device in performing the DMA transfer. This service allows for scatter/gather capability of a device and also allows the device to combine multiple requests that are contiguous with respect to the device. The lists are passed via the **dio** structure. If the **d_map_list** service is unable to complete the mapping due to exhausting the capacity of the provided **dio** structure, the **DMA_DIOFULL** error is returned. If the **d_map_list** service is unable to complete the mapping due to exhausting resources required for the mapping, the **DMA_NORES** error is returned. In both of these cases, the *bytes_done* field of the **dio** virtual list is set to the number of bytes successfully mapped. This byte count is a multiple of the *minxfer* size for the device as provided on the call to **d_map_list**. The *resid_iov* field is set to the index of the remaining *d_iovec* fields in the list. Unless the **DMA_BYPASS** flag is set, this service verifies access permissions to each page. If an access violation is encountered on a page with the list, the **DMA_NOACC** error is returned, and the *bytes_done* field is set to the number of bytes preceding the faulting *iovec*. If the mapping is for short term (that is, it is unmapped as soon as the I/O is complete), you must set the **DMA_STMAP** flag.

Note:

1. When the **DMA_NOACC** return value is received, no mapping is done, and the bus list is undefined. In this case, the *resid_iov* field is set to the index of the *d_iovec* that encountered the access violation.

2. You can use the **D_MAP_LIST** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_map_list** kernel service.

Return Values

Item	Description
DMA_NORES	Indicates that resources were exhausted during mapping.

Note: **d_map_list** possible partial transfer was mapped. Device driver may continue with partial transfer and submit the remainer on a subsequent **d_map_list** call, or call **d_unmap_list** to undo the partial mapping. If a partial transfer is issued, then the driver must call **d_unmap_list** when the I/O is complete.

Item	Description
DMA_DIOFULL	Indicates that the target bus list is full.

Note: **d_map_list** possible partial transfer was mapped. Device driver may continue with partial transfer and submit the remainder on a subsequent **d_map_list** call, or call **d_unmap_list** to undo the partial mapping. If a partial transfer is issued, then the driver must call **d_unmap_list** when the I/O is complete.

Item	Description
DMA_NOACC	Indicates no access permission to a page in the list.

.

Note: **d_map_list** no mapping was performed. No need for the device driver to call **d_unmap_list**, but the driver must fail the faulting I/O request, and resubmit any remainder in a subsequent **d_map_list** call.

Item	Description
DMA_SUCC	Indicates that the entire transfer successfully mapped.

Note: **d_map_list** successful mapping was performed. Device driver must call **d_unmap_list** when the I/O is complete. In the case of a long-term mapping, the driver must call **d_unmap_list** when the long-term mapping is no longer needed.

Related reference:

"d_map_init Kernel Service" on page 81

"d_map_init_ext Kernel Service" on page 82

d_map_page Kernel Service

Purpose

Performs platform-specific DMA mapping for a single page.

Syntax

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

int d_map_page(*handle, flags, baddr, *busaddr, *xmp)
struct d_handle *handle;
int flags;
caddr_t baddr;
uint *busaddr;
struct xmem *xmp;
```

Note: The following is the interface definition for **d_map_page** when the **DMA_ADDRESS_64** and **DMA_ENABLE_64** flags are set on the **d_map_init** call.

```
int d_map_page(*handle, flags, baddr, *busaddr, *xmp)
struct d_handle *handle;
int flags;
unsigned long long baddr;
unsigned long long *busaddr;
struct xmem *xmp;
```

Parameters

Item	Description
handle	Indicates the unique handle returned by the **d_map_init** kernel service.
flags	Specifies one of the following flags: **DMA_READ** Transfers from a device to memory. **BUS_DMA** Transfers from one device to another device. **DMA_BYPASS** Do not check page access. **DMA_STMAP** Indicates a short-term mapping.
baddr	Specifies the buffer address.
busaddr	Points to the *busaddr* field.
xmp	Cross-memory descriptor for the buffer.

Description

The **d_map_page** kernel service is a bus-specific utility routine determined by the **d_map_init** or **d_map_init_ext** kernel service that performs platform specific mapping of a single 4KB or less transfer for DMA master devices. The **d_map_page** kernel service is a fast-path version of the **d_map_list** service. The entire transfer amount must fit within a single page in order to use this service. This service accepts a virtual address and completes the appropriate bus address for the device to use in the DMA transfer. Unless the **DMA_BYPASS** flag is set, this service also verifies access permissions to the page. If the mapping is for short term (that is, it is unmapped as soon as the I/O is complete), you must set the **DMA_STMAP** flag.

If the buffer is a global kernel space buffer, the cross-memory descriptor can be set to point to the exported **GLOBAL** cross-memory descriptor, *xmem_global*.

If the transfer is unable to be mapped due to resource restrictions, the **d_map_page** service returns **DMA_NORES**. If the transfer is unable to be mapped due to page access violations, this service returns **DMA_NOACC**.

Note: You can use the **D_MAP_PAGE** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_map_page** kernel service.

Return Values

Item	Description
DMA_NORES	Indicates that resources are unavailable.

Note: **d_map_page** no mapping is done, device driver must wait until resources are freed and attempt the **d_map_page** call again.

Item	Description
DMA_NOACC	Indicates no access permission to the page.

Note: **d_map_page** no mapping is done, device driver must fail the corresponding I/O request.

Item	Description
DMA_SUCC	Indicates that the *busaddr* parameter contains the bus address to use for the device transfer.

Note: **d_map_page** successful mapping was done, device driver must call **d_unmap_page** when I/O is complete, or when device driver is finished with the mapped area in the case of a long-term mapping.

Related reference:

"d_alloc_dmamem Kernel Service" on page 60

"d_map_init Kernel Service" on page 81

"d_map_list Kernel Service" on page 83

d_map_query Kernel Service

Purpose

Queries the amount of direct memory access (DMA) space or DMA windows available on the partition end point. To use full 64-bit DMA all device drivers must call the **d_map_query** kernel service before attempting to initialize a new DMA window or before attempting to allocate DMA space within an existing DMA window by using the **d_map_init_ext** kernel service. Device drivers that do not use full 64-bit DMA should not call this service.

Syntax

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

kerrno_t d_map_query(bid, slot, flags, cmd, dq_info)
uint64_t  bid;
uint64_t  slot;
uint64_t  flags;
uint64_t  cmd;
void *    dq_info;
```

Parameter

Item	Description
bid	Specifies the bus identifier.
slot	Specifies the slot on the parent bus. This is the same as the **connwhere** property in the **CuDv** object class for the device.
flags	Specifies flags for the **d_map_query** kernel service. For future support, this parameter must be set to 0.
cmd	Specifies the type of query that the **d_map_query** kernel service will execute.
dq_info	Specifies the **dq_ddw_resources_t** or **dq_dma_available_t** structure based on which *cmd* parameter was defined.

Description

The **d_map_query** kernel service allows the device driver to determine the amount of DMA space available within the DMA window or the amount of DMA windows available for a particular partition end point.

The **d_map_query** kernel service Dynamic DMA Windows Query (**DDW_QUERY**) option is supported only on partition end points that support the dynamic DMA windows (**DDW_QUERY**) option. The **d_map_query** kernel service can also be used to determine the dynamic DMA windows capability of a particular partition endpoint. When a slot is initialized on a reboot or power-on operation of the partition or on a DR isolate operation that encompasses the partition endpoint, a default DMA window is always allocated for less than 4 GB. After the first call to the **d_map_query** kernel service with the **DDW_QUERY** option, the default DMA window is removed. This leaves no usable DMA window on the partition endpoint until the **d_map_init_ext** kernel service is called to initialize a new DMA window.

Note: The **DDW_QUERY** option should only be used by device drivers that fully support the 64-bit DMA.

The caller of the **d_map_query** kernel service must pass the desired command to the *cmd* parameter and have the appropriate *dq_info* parameter initialized.

The options available for the *cmd* parameter are defined in the **<sys/dma.h>** header file, and are described as follows:

DDW_QUERY
Returns the number of additional DMA windows available for a partition endpoint. The **dq_ddw_resources_t** structure must be passed to the *dq_info* parameter for this command. The **dqdr_version** field in the structure should be assigned as **DQDR_VERSION**.

DMA_QUERY
Returns the maximum amount of contiguous pages available for a given page size in all existing DMA windows. The **dq_dma_available_t** structure must be passed to the **dq_info** parameter for this command. The **dqda_version** field in the structure should be assigned as **DQDA_VERSION** and the corresponding I/O page size for the query must be specified. The supported I/O page size for the DMA operation can be obtained from the **d_map_query** kernel service by running the **DDW_QUERY** command.

The **dq_ddw_resources_t** and **dq_dma_available_t** structures are defined in the **<sys/dma.h>** as follows:

```
typedef struct dq_ddw_resources
 {   /* input by caller */
       uint64_t dqdr_version;
       /* returned to caller */
   uint64_t dqdr_supported_page_sizes;
   uint64_t dqdr_windows_avail;
       /* Amount of dynamic DMA windows available.
    * If DDW_QUERY, is not available
    * 0 will be returned.
       */
        uint64_t dqdr_max_pages; /* Largest number of contiguous pages available.*/
      uint64_t dqdr_rsvd1;          /* reserved for future use */
      uint64_t dqdr_rsvd2;           /* reserved for future use */
      uint64_t dqdr_rsvd3;           /* reserved for future use */
      uint64_t dqdr_rsvd4;           /* reserved for future use */
 } dq_ddw_resources_t;
 /*
  * The dq_dma_available structure is to be used in d_map_querywith the
  * DMA_QUERY cmd specified
  */
 typedef struct dq_dma_available
 {
      /*input by caller */
      uint64_t dqda_version;
      uint64_t dqda_io_page_size;
      /* Page size in bytes, should only be equal to the supported pagesize */
      /* returned to caller for DMA_Query*/
      uint64_t dqda_pages_available;
  uint64_t dqda_rsvd1;          /* reserved for future use */
  uint64_t dqda_rsvd2;          /* reserved for future use */
  uint64_t dqda_rsvd3;          /* reserved for future use */
  uint64_t dqda_rsvd4;          /* reserved for future use */
 } dq_dma_available_t;
```

Execution Environment

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

Return Values

Item	Description
0	Success
Kerrno	Error occurred

Related reference:
"d_map_clear Kernel Service" on page 79
"d_map_list Kernel Service" on page 83
"d_unmap_list Kernel Service" on page 106

d_map_slave Kernel Service

Purpose

Accepts a list of virtual addresses and sizes and sets up the slave DMA controller.

Syntax

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

int d_map_slave (*handle, flags, minxfer, *vlist, chan_flag)
struct d_handle *handle;
int flags;
int minxfer;
struct dio *vlist;
uint chan_flag;
```

Parameters

Item	Description
handle	Indicates the unique handle returned by the **d_map_init** kernel service.
flags	Specifies one of the following flags: **DMA_READ** Transfers from a device to memory. **BUS_DMA** Transfers from one device to another device. **DMA_BYPASS** Do not check page access.
minxfer	Specifies the minimum transfer size for the device.
vlist	Specifies a list of buffer addresses and lengths.
chan_flag	Specifies the device and bus specific flags for the transfer.

Description

The **d_map_slave** kernel service accepts a list of virtual buffer addresses and sizes and sets up the slave DMA controller for the requested DMA transfer. This includes setting up the system address generation hardware for a specific slave channel to indicate the specified data buffers, and enabling the specific hardware channel. The **d_map_slave** kernel service is not an exported kernel service, but a bus-specific utility routine determined by the **d_map_init** kernel service and provided to the caller through the **d_handle** structure.

This service allows for scatter/gather capability of the slave DMA controller and also allows the device driver to coalesce multiple requests that are contiguous with respect to the device. The list is passed with the **dio** structure. If the **d_map_slave** kernel service is unable to complete the mapping due to resource, an error, **DMA_NORES** is returned, and the **bytes_done** field of the **dio** list is set to the number of bytes that were successfully mapped. This byte count is guaranteed to be a multiple of the *minxfer* parameter size of the device as provided to **d_map_slave**. Also, the *resid_iov* field is set to the index of the remaining *d_iovec* that could not be mapped. Unless the **DMA_BYPASS** flag is set, this service will verify access permissions to each page. If an access violation is encountered on a page within the list, an error, **DMA_NOACC** is returned and no mapping is done. The *bytes_done* field of the virtual list is set to the number of bytes preceding the faulting *iovec*. Also in this case, the *resid_iov* field is set to the index of the *d_iovec* entry that encountered the access violation.

The virtual addresses provided in the *vlist* parameter can be within multiple address spaces, distinguished by the cross-memory structure pointed to for each element of the **dio** list. Each cross-memory pointer can point to the same cross-memory descriptor for multiple buffers in the same address space, and for global space buffers, the pointers can be set to the address of the exported GLOBAL cross-memory descriptor, *xmem_global*.

The *minxfer* parameter specifies the absolute minimum data transfer supported by the device(the device blocking factor). If the device supports a minimum transfer of 512 bytes (floppy and disks, for example), the *minxfer* parameter would be set to 512. This allows the underlying services to map partial transfers to a correct multiple of the device block size.

Note:

1. The **d_map_slave** kernel service does not support more than one outstanding DMA transfer per channel. Attempts to do multiple slave mappings on a single channel will corrupt the previous mappings.
2. You can use the **D_MAP_SLAVE** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_map_clear** kernel service.
3. The possible flag values for the *chan_flag* parameter can be found in **/usr/include/sys/dma.h**. These flags can be logically ORed together to reflect the desired characteristics of the device and channel.
4. If the **CH_AUTOINIT** flag is used then the transfer described by the **vlist** pointer is limited to a single buffer address with a length no greater than 4K bytes.

Return Values

Item	Description
DMA_NORES	Indicates that resources were exhausted during the mapping.
DMA_NOACC	Indicates no access permission to a page in the list.
DMA_BAD_MODE	Indicates that the mode specified by the *chan_flag* parameter is not supported.

Related reference:

"d_map_init Kernel Service" on page 81

dmp_add Kernel Service

Purpose

Specifies data to be included in a system dump by adding an entry to the master dump table. Callers should use the "dmp_ctl Kernel Service" on page 95. This service is provided for compatibility purposes.

Syntax

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

int dmp_add
( cdt_func)
struct cdt * ( (*cdt_func) (  ));
```

Description

Kernel extensions use the **dmp_add** service to register data areas to be included in a system dump. The **dmp_add** service adds an entry to the master dump table. A master dump table entry is a pointer to a function provided by the kernel extension that will be called by the kernel dump routine when a system dump occurs. The function must return a pointer to a component dump table structure.

When a dump occurs, the kernel dump routine calls the function specified by the *cdt_func* parameter twice. On the first call, an argument of 1 indicates that the kernel dump routine is starting to dump the data specified by the component dump table. On the second call, an argument of 2 indicates that the kernel dump routine has finished dumping the data specified by the component dump table. Kernel extensions should allocate and pin their component dump tables and call the **dmp_add** service during initialization. The entries in the component dump table can be filled in later. The *cdt_func* routine must not attempt to allocate memory when it is called.

Note: In AIX Version 7.1, this function automatically serializes CDT functions with I/O during dump time. The need for this function is device specific. Only the developer of the device can determine if this routine needs to be used. It is only recommended for devices that can be on the dump I/O path. Serializing I/O during dump time might degrade dump performance. Devices that are not on the dump path must either use the **dmp_ctl** routine or the RAS system dump interface.

The Component Dump Table

The component dump table structure specifies memory areas to be included in the system dump. The structure type (**struct cdt**) is defined in the **/usr/include/sys/dump.h** file. A **cdt** structure consists of a fixed-length header (**cdt_head** structure) and an array of one or more **cdt_entry** structures. The **cdt_head** structure contains a component name field, which should be filled in with the name of the kernel extension, and the length of the component dump table. Each **cdt_entry** structure describes a contiguous data area, giving a pointer to the data area, its length, a segment register, and a name for the data area.

Use of the Formatting Routine

Each kernel extension that includes data in the system dump can install a unique formatting routine in the **/var/adm/ras/dmprtns** directory.The name of the formatting routine must match the component name field of the corresponding component dump table.

The dump image file includes a copy of each component dump table used to dump memory.A sample dump formatter is shipped with **bos.sysmgt.serv_aid** in the **/usr/samples/dumpfmt** directory.

Organization of the Dump Image File

Memory dumped for each kernel extension is laid out as follows in the dump image file. The component dump table is followed by a bitmap for the first data area, then the first data area itself, then a bitmap for the next data area, the next data area itself, and so on.

The bitmap for a specific data area indicates which pages of the data area are 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 bitmap 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 bitmap is provided in the `/usr/include/sys/dump.h` file.

Parameters

Item	**Description**
cdt_func	Specifies a function that returns a pointer to a component dump table entry. The function and the component dump table entry both must reside in pinned global memory.

Execution Environment

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

Return Values

Item	**Description**
0	Indicates a successful operation.
-1	Indicates that the function pointer to be added is already present in the master dump table.

Related reference:
"dmp_ctl Kernel Service" on page 95
Related information:
exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine
RAS Kernel Services

dmp_compspec and dmp_compext Kernel Services

Purpose

Specifies a component and callback parameters to be included in the dump.

Syntax

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

kerrno_t dmp_compspec (flags, comp, anchor, extid, p1, p2, ..., NULL)
long flags;
long comp;
void *anchor;
dmp_extid_t *extid;
```

```
char *p1;
char *p2;
...

kerrno_t dmp_compext (extid, p1, p2, ..., NULL)
dmp_extid_t extid;
char *p1;
char *p2;
...
```

Parameters

Item	Description
anchor	Points to the associated **ldmp_parms_t** data structure or to an **ldmp_prepare_t** data structure.
comp	Specifies the component, specified as indicated by the flags.
extid	Points to an item of **dmp_extid_t** type, for the **dmp_compspec** kernel service, where an identifier is returned, if you use the **dmp_compext** kernel service to provide additional parameters for the component being dumped. This identifier might then be specified to add additional parameters to the component using the **dmp_compext** kernel service. The *extid* parameter can be NULL.
flags	You can specify the following values:

DCF_FAILING
Indicates that this is the failing component. You can only specify one failing component.

DCF_FIRST
Indicates that this component is to be dumped first. Normally components are dumped in the order specified.

Note:

- The **DCF_FIRST** value is only valid when the anchor refers to an **ldmp_parms_t** data item. It is not valid when the callback receives the **RASCD_LDMP_PREPARE** command.
- The last component specified to be dumped first is the one dumped first.

DCF_LEVEL0 - DCF_LEVEL9
Indicates the detail level, 0 through 9, to dump this component. If none of these flags are set, the component is dumped at its current level.

DCF_MINIMAL
Indicates the DCF_LEVEL1 level.

DCF_NORMAL
Indicates the DCF_LEVEL3 level.

DCF_DETAIL
Indicates the DCF_LEVEL7 level.

DCF_LONG
Indicates that the parameters are two parameters of long type. Rather than passing in an unlimited number of strings, a component can be passed in two long data items, as in the case with pseudo-components.

One and only one of the following component specification flags must be given. They specify how the component is specified in the dc_component field:

DCF_BYPNAME
Indicates that the component is specified by path name.

DCF_BYLNAME
Indicates that the component is specified by logical alias.

DCF_BYTYPE
Indicates that the component is specified by type.

DCF_BYCB
Indicates that the component is specified by ras_block_t.

Item	Description
p1, p2 ...	Specifies the component's parameters, the last of which must be NULL. If keyword parameters are being specified, The parameters must be strings, and contain the keyword and its values. If multiple keyword and value pairs appear in a single parameter, they are separated with blanks. For example, the *p1* parameter can be `foo=1234`, and the *p2* parameter can be `bar=5678,16`. Also, the *p1* parameter can be `foo=1234 bar=5678`. If the **DCF_LONG** flag is set, two parameters of long type are passed in. In this case, the *p1* and *p2* parameters contain the values of long type, and no more parameters can be specified.

Description

The **dmp_compspec** and **dmp_compext** kernel services provide components and their callback parameters for a dump. You can only use these kernel services in a live dump.

The **dmp_compspec** kernel service is used before you start a live dump with the **livedump** kernel service. You can also use this kernel service when a component's callback wants to include another component in a live dump, that is, when the callback receives the **RASCD_LDMP_PREPARE** command.

Multiple components can be included in a live dump.

The **dmp_compext** function is used to provide additional parameters for a component.

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_RAS_DMP_COMPSPEC_FLAGS	Indicates that the flags specification is not valid.
EINVAL_RAS_DMP_COMPSPEC_COMP	Indicates that the component specification is not valid.
EINVAL_RAS_DMP_COMPSPEC_NOTAWARE	Indicates that the specified component must support live dump.
EINVAL_RAS_DMP_COMPSPEC_ANCHOR	Indicates that the anchor specification is not valid.
EFAULT_RAS_DMP_COMPSPEC_ANCHOR	Indicates that the storage the anchor parameter refers to is not valid.
EFAULT_RAS_DMP_COMPSPEC_EXTID	Indicates that the storage the *extid* parameter refers to is not valid.
EFAULT_RAS_DMP_COMPSPEC_PARMS	Indicates that a parameter address is not valid.
EINVAL_RAS_LDMP_ESTIMATE	Indicates that the anchor parameter indicates a dump size estimate request, but the **dmp_compspec** call was not made from the process environment.
EINVAL_RAS_DMP_COMPSPEC_NOADD	Indicates that components cannot be added to this dump, that is, the dump type flags, ldpr_flags, have the LDT_NOADD bit set.
EINVAL_RAS_DMP_COMPSPEC_FAILING	Indicates that the failing component has already been specified.
ENOMEM_RAS_DMP_COMPSPEC	Indicates that no storage is available.
EINVAL_RAS_DMP_COMPEXT_EXTID	Indicates that the *extid* parameter does not refer to a valid component.
EFAULT_RAS_DMP_COMPEXT_EXTID	Indicates that the storage the *extid* parameter refers to is not valid.
EFAULT_RAS_DMP_COMPEXT_PARMS	Indicates that the storage a parameter refers to is not valid.
EBUSY_RAS_DMP_COMPEXT	Indicates that the specification of this component is complete, and no more parameters can be added. This happens if the component the *extid* parameter referred to has already completed its **RASCD_LDMP_PREPARE** processing.
ENOMEM_RAS_DMP_COMPEXT	Indicates that no storage is available.

Related reference:

"livedump Kernel Service" on page 336

"ldmp_setupparms Kernel Service" on page 334

"ras_ret_query_parms Kernel Service" on page 433

dmp_ctl Kernel Service

Purpose

Adds and removes entries to the master dump table.

Syntax

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

     int dmp_ctl(op, parmp)
     int op;
     struct dmpctl_data *parmp;
```

Description

The **dmp_ctl** kernel service is used to manage dump routines. It replaces the **dmp_add** and **dmp_del** kernel services which are still supported for compatibility reasons. The major differences between routines added with the **dmp_add()** command and those added with the **dmp_ctl()** command are:

- The routines are invoked differently from routines added with the **dmp_add** kernel service. Routines added using the **dmp_ctl** kernel service return a void pointer, to a dump table or to a dump size estimate.
- Routines added with the **dmp_ctl** kernel service are expected to ignore functions they don't support. For example, they should not trap if they receive an unrecognized request. This allows future functionality to be added without all users needing to change.

The **dmp_ctl** kernel service is used to request that an amount of memory be set aside in a global buffer. This will then be used by the routine to store data not resident in memory. An example of such data is dump data provided by an adapter. Without a global buffer, the data would need to be placed into a pinned buffer allocated at configuration time. Each component would need to allocate its own pinned buffer.

The system dump facility maintains a global buffer for such data. This buffer is allocated when it is first requested, with the requested size. Another dump routine requesting more data causes the buffer to be reallocated with the larger size. Since this buffer must be maintained in pinned storage for the life of the system, only ask for as much memory as is required. Asking for an excessive amount of storage will compromise system performance by reserving too much pinned storage.

Any dump routine using the global buffer is called whenever dump data is required. Routines are only called once to provide such data. Their dump table addresses are saved and used if the dump is restarted.

Note: The **dmp_ctl** kernel service can also be used by a dump routine to report a routine failure. This may be necessary if the routine detects that it can't dump what needs to be dumped for some reason such as corruption of a data structure.

Note: Beginning with AIX Version 6.1 with the 6100-02 Technology Level, the **dmp_ctl** kernel service supports that DMPFUNC_SERIALIO operation flag.

Dump Tables

A dump routine returns a component dump table that begins with DMP_MAGIC, which is the magic number for the 32- or 64-bit dump table. If the unlimited sized dump table is used, the magic number is DMP_MAGIC_U and the **cdt_u** structure is used. If this is the case, the dump routine is called repeatedly until it returns a null **cdt_u** pointer. The purpose of the unlimited size dump table is to provide a way to dump an unknown number of data areas without having to preallocate the largest possible array of

cdt_entry elements as is required for the classic dump table. The definitions for dump tables are in the **sys/dump.h** include file.

Parameters

dmp_ctl operations and the **dmpctl_data** structure are defined in the **dump.h** text file.

<table>
<tr><th>Item</th><th>Description</th></tr>
<tr><td>op</td><td>Specifies the operation to perform.</td></tr>
<tr><td>parmp</td><td>Points to a <b>dmpctl_data</b> structure containing values for the specified operation. The <b>dmpctl_data</b> structure is defined in the <b>/usr/include/sys/dump.h</b> file as follows:
<pre>
/* Dump Routine failures data. */
struct __rtnf {
       int rv;   /* error code. */
       ulong vaddr;   /* address. */
       vmhandle_t handle;   /* handle */
};

typedef   void *((*__CDTFUNCENH)(int op, void *buf));
struct dmpctl_data {
       int dmpc_magic;   /* magic number   */
       int dmpc_flags;   /* dump   routine   flags. */
       __CDTFUNCENH dmpc_func;
       union {
              u_longlong_t bsize;   /* Global buffer size requested. */
              struct __rtnf rtnf;
       } dmpc_u;
};
#define   DMPC_MAGIC1 0xdcdcdc01
#define   DMPC_MAGIC DMPC_MAGIC1
#define   dmpc_bsize dmpc_u.bsize
#define dmpcf_rv dmpc_u.rtnf.rv
#define dmpcf_vaddr dmpc_u.rtnf.vaddr
#define dmpcf_handle dmpc_u.rtnf.handle
</pre></td></tr>
</table>

The supported operations and their associated data are:

Item	Description
DMPCTL_ADD	Adds the specified dump routine to the master dump table. This requires a pointer to the function and function type flags. Supported type flags are: **DMPFUNC_CALL_ON_RESTART** Calls this function again if the dump is restarted. A dump function is only called once to provide dump data. If the function must be called and the dump is restarted on the secondary dump device, then this flag must be set. The **DMPFUNC_CALL_ON_RESTART** flag must be set if this function uses the global dump buffer. It also must be set if the function uses an unlimited size dump table, a table with DMP_MAGIC_U as the magic number. **DMPFUNC_GLOBAL_BUFFER** This function uses the global dump buffer. The size is specified using the **dmpc_bsize** field. **DMPFUNC_SERIALIO** Enables serialized I/O during dump time. The need for this flag is device specific. Only the developer of the device can determine if this flag needs to be set. It is only recommended for devices that can be on the dump I/O path. Serializing I/O during dump time can degrade dump performance. The default, without this flag, is to allow I/O to occur in parallel with CDT function calls.
DMPCTL_DEL	Deletes the specified dump function from the master dump table.
DMPCTL_RTNFAILURE	Reports an inability to dump required data. The routine must set the dmpc_func, dmpcf_rV, dmpcf_vaddr, and dmpcf_handle fields.

Dump function invocation parameters:

Item	Description
operation code	Specifies the operation the routine is to perform. Operation codes are: **DMPRTN_START** The dump is starting for this dump table. Provide data. **DMPRTN_DONE** The dump is finished. This call is provided so that a dump routine can do any cleanup required after a dump. This is specific to a device for which information was gathered. It does not free memory, since such memory must be allocated before the dump is taken. **DMPRTN_AGAIN** Provide more data for this unlimited dump table. The routine must have first passed back a dump table beginning with DMP_MAGIC_U. When finished, the function must return a NULL. **DMPRTN_ESTIMATE** Provide a size estimate. The function must return a pointer to an item of type dmp_sizeest_t. See the examples later in this article.
buffer pointer	This is a pointer to the global buffer, or NULL if no global buffer space was requested.

Return Values

Item	Description
0	Returned if successful.
EINVAL	Returned if one or more parameter values are invalid.
ENOMEM	Returned if the global buffer request can't be satisfied.
EEXIST	Returned if the dump function has already been added.

Examples

1. To add a dump routine (dmprtn) that can be called once to provide data, type:

```
void *dmprtn(int op, void *buf);
          struct cdt cdt;
          dmp_sizeest_t estimate;

          config()
          {
                  struct dmpctl_data parm;
                  ...

                  parm.dmpc_magic = DMPC_MAGIC1;
                  parm.dmpc_func = dmprtn;
                  parm.dmpc_flags = 0;
                  ret = dmp_ctl(DMPCTL_ADD, &parm);

                  ...
          }

          /*
           * Dump routine.
           *
           * input:
           *   op - dump routine operation.
           *   buf - NULL since no global buffer is used.
           *
           * returns:
           *   A pointer to the component dump table.
           */
          void *
          dmprtn(int op, void *buf)
          {
                  void *ret;
```

```
        switch(op) {
        case DMPRTN_START: /* Provide dump data. */
                ...
                ret = (void *)&cdt;
                break;
        case DMPRTN_ESTIMATE:
                ret = (void *)&estimate;
                break;
        default:
                        break;
        }

        return(ret);
}
```

2. To add a dump routine (dmprtn) that requests 16 kb of global buffer space, type:

...

```
#define BSIZ 16*1024
dmp_sizeest_t estimate;

config()
{
        ...
        parm.dmpc_magic = DMPC_MAGIC1;
        parm.dmpc_func = dmprtn;
        parm.dmpc_flags = DMPFUNC_CALL_ON_RESTART|DMPC_GLOBAL_BUFFER;
        parm.dmpc_bsize = BSIZ;
        ret = dmp_ctl(DMPCTL_ADD, &parm);
        ...
}

/*
 * Dump routine.
 *
 * input:
 *   op - dump routine operation.
 *   buf - points to the global buffer.
 *
 * output:
 *   Return a pointer to the dump table or to the estimate.
 */
void *
dmprtn(int op, void *buf)
{
        void *ret;

        switch(op) {
        case DMPRTN_START: /* Provide dump data. */
                ...
                (Put data in buffer at buf.)
                ret = (void *)&cdt;
                break;
        case DMPRTN_ESTIMATE:
                ret = (void *)&estimate;
                break;
        default:
                        break;
        }

        return(ret);
}
```

Related reference:

“dmp_del Kernel Service” on page 99

Related information:

Dump Special File

System Dump Facility

dmp_del Kernel Service

Purpose

Deletes an entry from the master dump table. Callers should use the “dmp_ctl Kernel Service” on page 95. This service is provided for compatibility purposes.

Syntax

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

dmp_del ( cdt_func_ptr)
struct cdt * ( (*cdt_func_ptr) (  ));
```

Description

Kernel extensions use the **dmp_del** kernel service to unregister data areas previously registered for inclusion in a system dump. A kernel extension that uses the “dmp_add Kernel Service” on page 91 to register such a data area can use the **dmp_del** service to remove this entry from the master dump table.

Parameters

Item	Description
cdt_func_ptr	Specifies a function that returns a pointer to a component dump table. The function and the component dump table must both reside in pinned global memory.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.
-1	Indicates that the function pointer to be deleted is not in the master dump table.

Related reference:

“dmp_add Kernel Service” on page 91

“dmp_ctl Kernel Service” on page 95

Related information:

RAS Kernel Services

dmp_eaddr, dmp_context, dmp_tid, dmp_pid, dmp_errbuf, dmp_mtrc, dmp_systrace, and dmp_ct Kernel Services

Purpose

Provides functions for common dump tasks.

Syntax

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

```
kerrno_t dmp_eaddr (flags, anchor, name, addr, sz)
long flags;
void *anchor;
char *name;
long addr;
long sz;

kerrno_t dmp_context (flags, anchor, name, ctx_type, p2)
long flags;
void *anchor;
char *name;
long ctx_type;
long p2;

kerrno_t dmp_tid (flags, anchor, name, tid, unused)
long flags;
void *anchor;
char *name;
tid_t tid;
void *unused;

kerrno_t dmp_pid (flags, anchor, name, pid, unused)
long flags;
void *anchor;
char *name;
pid_t pid;
void *unused;

kerrno_t dmp_errbuf (flags, anchor, name, erridx, unused)
long flags;
void *anchor;
char *name;
ulong erridx;
long unused;

kerrno_t dmp_mtrc (flags, anchor, name, com_sz, rare_sz)
long flags;
void *anchor;
char *name;
size_t com_sz;
size_t rare_sz;

kerrno_t dmp_systrace (flags, anchor, name, sz, unused)
long flags;
void *anchor;
char *name;
long sz;
long unused;

kerrno_t dmp_ct (flags, anchor, name, rasb, sz)
long flags;
void *anchor;
char *name;
ras_block_t rasb;
size_t sz;
```

Parameters

Item	Description
anchor	Points to the associated **ldmp_parms_t** data structure or to an **ldmp_prepare_t** data structure.
flags	The *flags* parameter can be one or more of the following values: **DCF_FIRST** Specifies that this component is to be dumped first. Normally components are dumped in the order specified. **Note:** The last component specified to be dumped first is the one dumped first. **DCF_LEVEL0 - DCF_LEVEL9** Dumps the component at the specified detail level, 0 through 9. If none of these flags are set, the component is dumped at CD_LVL_NORMAL, detail level 3.
name	Specifies the name of the pseudo-component's dump table in the dump. The *name* parameter is only valid for the **dmp_eaddr** kernel service. You must specify the *name* parameter to NULL for the **dmp_context**, **dmp_tid**, **dmp_pid**, **dmp_errbuf**, **dmp_mtrc**, **dmp_systrace**, and **dmp_ct** kernel services.
unused	You must specify this parameter to NULL or 0.
The remaining parameters are pseudo-component dependent: **dmp_eaddr** *addr*	Specifies the effective address of the memory to be dumped.
sz	Specifies the length of the memory in bytes.
dmp_context *ctx_type*	Specifies the context to dump. It can be one of the following values: **DMP_CTX_CUR** To dump the current context. **DMP_CTX_PREV** To dump the previous context. **DMP_CTX_SPEC** To dump the context specified by the *p2* parameter. The *p2* parameter must contain the address of the **ksmtsave** structure for the context. **DMP_CTX_RWA** To dump the context from the supplied recovery work area. The *p2* parameter must contain the address of the recovery work area, rwa. **DMP_CTX_BID or DMP_CTX_LCPUID** To dump the context for the processor specified by the *p2* parameter. You can specify the processor either by the bind ID or by the logical ID. **DMP_CTX_TID** To dump the context of the thread specified by the *p2* parameter, which must contain the thread ID.
p2	Specifies the address of the context, the logical processor ID, the bind ID, or the thread ID dependent on the value of the *ctx_type* parameter.
dmp_tid *tid*	Specifies the ID of the thread to dump.
dmp_pid *pid*	Specifies the ID of the process to dump.
dmp_errbuf *erridx*	Specifies the kernel workload partition (WPAR) ID of the partition's error logging buffer to dump. The value of 0 stands for the global buffer.
dmp_mtrc *com_sz*	Specifies the amount of common to dump.
rare_sz	Specifies the amount of rare data to dump.
dmp_systrace *sz*	Specifies the amount of system trace data to dump.If the *sz* parameter is set to 0, all the buffered trace data is dumped, up to the amount allowed by the detail level.
dmp_ct *rasb*	Specifies the ras_block_t of the component whose component trace is to be dumped.
sz	Specifies the amount of data to dump. If the *sz* parameter is set to 0, all the components' trace data is dumped, up to the limit for the detail level.

Description

The **dmp_eaddr** kernel service dumps memory by effective address.

The **dmp_context** kernel service dumps the specified thread context.

The **dmp_tid** kernel service dumps the kernel data for a thread.

The **dmp_pid** kernel service dumps the kernel data for a process.

The **dmp_errbuf** kernel service dumps the error logging buffer for the specified partition.

The **dmp_mtrc** kernel service dumps entries from the lightweight memory trace buffers.

The **dmp_systrace** dumps entries from the system trace buffers.

The **dmp_ct** dumps component trace entries.

Execution Environment

The **dmp_eaddr**, **dmp_context**, **dmp_tid**, **dmp_pid**, **dmp_errbuf**, **dmp_mtrc**, **dmp_systrace**, and **dmp_ct** kernel services can be called from either the process or interrupt environment.

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_DMP_PSEUDO	Indicates that the name parameter is not valid.
EINVAL_DMP_CHECK_ANCHOR	Indicates that no anchor was specified, or the anchor parameter does not point to an area of **ldmp_parms_t** or **ldmp_prepare_t** type.
EFAULT_DMP_CHECK_ANCHOR	Indicates that the storage specified by the *anchor* parameter is not valid.
EINVAL_RAS_DMP_COMPSPEC_FLAGS	Indicates that the flags specification is not valid. This error also occurs if the **DCF_FIRST** flag is specified when the anchor is an **ldmp_prepare_t** data item.
EINVAL_RAS_DMP_COMPSPEC_NOADD	Indicates that components cannot be added to this dump.
ENOMEM_RAS_DMP_COMPSPEC	Indicates that the storage is not sufficient.
EINVAL_RAS_DMP_EADDR	Indicates that the flags parameter is not valid.
EINVAL_RAS_DMP_CONTEXT	Indicates that the parameter of the **dmp_context** kernel service is not valid. This is also returned if the *p2* parameter is not used, but is not NULL.
ENOENT_RAS_DMP_CONTEXT_CTX_NOTFOUND	Indicates that the specified context was not found.
EFAULT_RAS_DMP_CONTEXT	Indicates that the storage the specified context pointer points to is not valid.
EINVAL_RAS_DMP_TID	Indicates that the parameter of the **dmp_tid** kernel service is not valid.
EINVAL_RAS_DMP_PID	Indicates that the parameter of the **dmp_pid** kernel service is not valid.
EINVAL_RAS_DMP_ERRBUF	Indicates that the parameter of the **dmp_errbuf** kernel service is not valid.
ECHRNG_RAS_DMP_ERRBUF	Indicates that the *erridx* parameter is out of range.
EINVAL_RAS_DMP_MTRC	Indicates that the parameter of the **dmp_mtrc** kernel service is not valid.
ENOENT_RAS_DMP_MTRC	Indicates that the lightweight memory trace is not active.
EINVAL_RAS_DMP_SYSTRACE	Indicates that the parameter of the **dmp_systrace** kernel service is not valid.
ENOENT_RAS_DMP_SYSTRACE	Indicates that the system trace is not active.
EINVAL_RAS_DMP_CT	Indicates that the parameter of the **dmp_ct** kernel service is not valid.
ENOMEM_RAS_DMP_CT	Indicates that the storage is not sufficient.
EINVAL_RAS_DMP_CT_GETPATH	Indicates that the specified component is not valid.
EINVAL_RAS_DMP_CT_LOOKUP	Indicates that an error occurred while this component was being validated.
ENOTSUP_RAS_DMP_CT	Indicates that the specified component does not have a component trace.

Related Information

The **livedump** kernel service and **dmp_kernext** kernel service.

Related reference:

"livedump Kernel Service" on page 336

"dmp_kernext Kernel Service" on page 103

dmp_kernext Kernel Service

Purpose

Causes the specified kernel extension to be shipped with the live dump for symbol resolution.

Syntax

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

kerrno_t dmp_kernext (anchor, ptr)void *anchor;
void *ptr;
```

Parameters

Item	Description
anchor	Points to either an **ldmp_parms_t** or **ldmp_prepare_t** structure.
ptr	Specifies an address within the kernel extension. If the value is 0, the dump includes information for all loaded kernel extensions.

Description

The **dmp_kernext** kernel service causes snap to package the specified kernel extension with the current live dump. This also includes loader information for the extension in the dump. You can specify the extension by setting the *ptr* parameter to a text or data address within the extension. The extension's file name is noted in the dump, and snap can be used to cause this file to be bundled with the snap data when the dump is collected for sending to IBM®.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_RAS_DMP_KERNEXT	Indicates that the *anchor* parameter is not valid.

Related reference:

"livedump Kernel Service" on page 336

Related information:

snap subroutine

d_roundup Kernel Service

Purpose

Rounds the value length up to a given number of cache lines.

Syntax

```
int  d_roundup(length)
int  length;
```

Parameter

Item	Description
length	Specifies the size in bytes to be rounded.

Description

To maintain cache consistency, buffers must occupy entire cache lines. The **d_roundup** service helps provide that function by rounding the value length up to a given number in integer form.

Execution Environment

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

Related reference:

"d_align Kernel Service" on page 60

"d_cflush Kernel Service" on page 61

Related information:

Understanding Direct Memory Access (DMA) Transfers

d_sync_mem Kernel Service

Purpose

Allows a device driver to indicate that previously mapped buffers may need to be refreshed.

Syntax

```
int d_sync_mem(d_handle_t handle, dio_t blist)
```

Description

The **d_sync_mem** service allows a device driver to indicate that previously mapped buffers may need to be refreshed, either because a new DMA is about to start or a previous DMA has now completed. **d_sync_mem** is not an exported kernel service, but a bus-specific utility determined by **d_map_init** based on platform characteristics and provided to the caller through the *d_handle* structure. **d_sync_mem** allows the driver to identify additional coherency points beyond those of the initial mapping (**d_map_list**) and termination of the mapping (**d_unmap_list**). Thus **d_sync_mem** provides a way to long-term map buffers and still handle potential data consistency problems.

The *blist* parameter is a pointer to the **dio** structure that describes the initial mapping, as returned by **d_map_list**. Note that for bounce buffering, the data direction is also implicitly defined by this initial mapping.

- If the **map_list** call describes a transfer from system memory to a device, subsequent **d_sync_mem** calls using the corresponding *blist* will synchronize the memory view. This assumes that the original system memory pages contain the correct data.
- If the **map_list** call describes a transfer from a device to system memory, then subsequent **d_sync_mem** calls will synchronize the memory view. This assumes that the bounce pages the device directly accessed contained the correct data.

Note: You can use the **D_SYNC_MEM** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_sync_mem** kernel service.

Parameters

Item	Description
d_handle_t	Indicates the unique dma handle returned by **d_map_init**.
dio_t blist	List of vectors returned by original **d_map_list**.

Return Values

Item	Description
DMA_SUCC	Buffers described by the *blist* have been synchronized.
DMA_FAIL	Buffers could not be synchronized.

Related reference:

"d_alloc_dmamem Kernel Service" on page 60

"d_map_list Kernel Service" on page 83

"d_unmap_list Kernel Service" on page 106

DTOM Macro for mbuf Kernel Services

Purpose

Converts an address anywhere within an **mbuf** structure to the head of that **mbuf** structure.

Syntax

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

DTOM ( bp);
```

Parameter

Item	Description
bp	Points to an address within an **mbuf** structure.

Description

The **DTOM** macro converts an address anywhere within an **mbuf** structure to the head of that **mbuf** structure. This macro is valid only for **mbuf** structures without an external buffer (that is, with the **M_EXT** flag not set).

This macro can be viewed as the opposite of the **MTOD** macro, which converts the address of an **mbuf** structure into the address of the actual data contained in the buffer. However, the **DTOM** macro is more general than this view implies. That is, the input parameter can point to any address within the **mbuf** structure, not merely the address of the actual data.

Example

The **DTOM** macro can be used as follows:

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

Related reference:

"MTOD Macro for mbuf Kernel Services" on page 375

Related information:

I/O Kernel Services

d_unmap_list Kernel Service

Purpose

Deallocates resources previously allocated on a **d_map_list** call.

Syntax

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

void d_unmap_list (*handle, *bus_list)
struct d_handle *handle
struct dio *bus_list
```

Note: The following is the interface definition for **d_unmap_list** when the **DMA_ADDRESS_64** and **DMA_ENABLE_64** flags are set on the **d_map_init** call.

```
void d_unmap_list (*handle,
*bus_list)
struct d_handle *handle;
struct dio_64 *bus_list;
```

Parameters

Item	Description
handle	Indicates the unique handle returned by the **d_map_init** kernel service.
bus_list	Specifies a list of bus addresses and lengths.

Description

The **d_unmap_list** kernel service is a bus-specific utility routine determined by the **d_map_init** kernel service that deallocates resources previously allocated on a **d_map_list** call.

The **d_unmap_list** kernel service must be called after I/O completion involving the area mapped by the prior **d_map_list** call. Some device drivers might choose to leave pages mapped for a long-term mapping of certain memory buffers. In this case, the driver must call **d_unmap_list** when it no longer needs the long-term mapping.

Note: You can use the **D_UNMAP_LIST** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_unmap_list** kernel service. If not, you must ensure that the **d_unmap_list** function pointer is non-**NULL** before attempting the call. Not all platforms require the unmapping service.

Related reference:

"d_map_init Kernel Service" on page 81

"d_map_list Kernel Service" on page 83

d_unmap_slave Kernel Service

Purpose

Deallocates resources previously allocated on a **d_map_slave** call.

Syntax

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

int d_unmap_slave (*handle)
struct d_handle *handle;
```

Parameters

Item	Description
handle	Indicates the unique handle returned by the **d_map_init** kernel service.

Description

The **d_unmap_slave** kernel service deallocates resources previously allocated on a **d_map_slave** call, disables the physical DMA channel, and returns error and status information following the DMA transfer. The **d_unmap_slave** kernel service is not an exported kernel service, but a bus-specific utility routine that is determined by the **d_map_init** kernel service and provided to the caller through the **d_handle** structure.

Note: You can use the **D_UNMAP_SLAVE** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_unmap_slave** kernel service. If not, you must ensure that the **d_unmap_slave** function pointer is non-**NULL** before attempting to call. No all platforms require the unmapping service.

The device driver must call **d_unmap_slave** when the I/O is complete involving a prior mapping by the **d_map_slave** kernel service.

Note: The **d_unmap_slave** kernel should be paired with a previous **d_map_slave** call. Multiple outstanding slave DMA transfers are not supported. This kernel service assumes that there is no DMA in progress on the affected channel and deallocates the current channel mapping.

Return Values

Item	Description
DMA_SUCC	Indicates successful transfer. The DMA controller did not report any errors and that the Terminal Count was reached.
DMA_TC_NOTREACHED	Indicates a successful partial transfer. The DMA controller reported the Terminal Count reached for the intended transfer as set up by the **d_map_slave** call. Block devices consider this an erro; however, for variable length devices this may not be an error.
DMA_FAIL	Indicates that the transfer failed. The DMA controller reported an error. The device driver assumes the transfer was unsuccessful.

Related reference:

"d_map_init Kernel Service" on page 81

d_unmap_page Kernel Service

Purpose

Deallocates resources previously allocated on a **d_unmap_page** call.

Syntax

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

void d_unmap_page (*handle, *busaddr)
struct d_handle *handle
uint *busaddr
```

Note: The following is the interface definition for **d_unmap_page** when the **DMA_ADDRESS_64** and **DMA_ENABLE_64** flags are set on the **d_map_init** call.

```
int d_unmap_page(*handle,
*busaddr)
struct d_handle *handle;
unsigned long long *busaddr;
```

Parameters

Item	Description
handle	Indicates the unique handle returned by the **d_map_init** kernel service.
busaddr	Points to the *busaddr* field.

Description

The **d_unmap_page** kernel service is a bus-specific utility routine determined by the **d_map_init** kernel service that deallocates resources previously allocated on a **d_map_page** call for a DMA master device.

The **d_unmap_page** service must be called after I/O completion involving the area mapped by the prior **d_map_page** call. Some device drivers might choose to leave pages mapped for a long-term mapping of certain memory buffers. In this case, the driver must call **d_unmap_page** when it no longer needs the long-term mapping.

Note: You can use the **D_UNMAP_PAGE** macro provided in the **/usr/include/sys/dma.h** file to code calls to the **d_unmap_page** kernel service. If not, you must ensure that the **d_unmap_page** function pointer is non-**NULL** before attempting the call. Not all platforms require the unmapping service.

Related reference:

"d_map_init Kernel Service" on page 81

dr_reconfig System Call

Purpose

Determines the nature of the DLPAR request.

Syntax

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

int dr_reconfig (flags, dr_info)
int flags;
dr_info_t *dr_info;
```

Description

The **dr_reconfig** system call is used by DLPAR-aware applications to adjust their use of resources in relation to a DLPAR request. Applications are notified about the usage through the **SIGRECONFIG** signal, which is generated three times for each DLPAR event.

The first time is to check with the application whether the DLPAR event should be continued. Using the **DR_EVENT_FAIL** flag, an application can indicate that the operation should be aborted, if it is not DLPAR-safe and its operation is considered vital to the system.

The application is notified the second time before the resource is added or removed, and the third time afterwards. Applications must attempt to control their scheduling priority and policy to guarantee timely delivery of signals. The system does not guarantee that every signal that is sent is delivered before advancing to the next step in the algorithm.

The **dr_reconfig** system call can also be used to notify applications about the changes to the workload partition that they are running. Applications are notified about changes to the CPU, memory capacity, and resources set.

The **dr_reconfig** interface is signal-handler safe and can be used by multi-threaded programs.

The **dr_info** structure is declared within the address space of the application. The kernel fills out data in this structure relative to the current DLPAR request. The user passes this structure identifying the current

DLPAR request, as a parameter to the kernel when the **DR_RECONFIG_DONE** flag is used. The **DR_RECONFIG_DONE** flag is used by the application to notify the kernel that necessary action to adjust their use of resources has been taken in response to the **SIGRECONFIG** signal sent to them. It is expected that the signal handler associated with the **SIGRECONFIG** signal calls the interface with the **DR_QUERY** flag to identify the phase of the DLPAR event, takes the appropriate action, and calls the interface with the **DR_RECONFIG_DONE** flag to indicate to the kernel that the signal has been handled. This type of acknowledgment to the kernel in each of the DLPAR phases enables a DLPAR event to perform efficiently.

With the addition of new fields to the **dr_info** structure, DR-aware applications can support the Micro-Partitioning® feature.

The *bindproc*, *softpset*, and *hardpset* bits are only set, if the request is to remove a cpu. If the *bindproc* is set, the process or one of its threads has a **bindprocessor** attachment, which must be resolved. If the *softpset* bit is set, the process has a Workload Manager (WLM) attachment, which can be changed by calling the appropriate WLM interface or by invoking the appropriate WLM command. If the *hardpset* bit is set, the appropriate **pset** API must be used.

Note: The *bcpu* and *lcpu* fields identify the cpu being removed and do not necessarily indicate that the process has a dependency that must be resolved. The *bindproc*, *softpset*, and *hardpset* bits are provided for that purpose.

The *plock* and *pshm* bits are only set, if the request is to remove memory and the process has **plock** memory or is attached to a pinned shared memory segment. If the *plock* bit is set, the process calls**plock** to unpin itself. If the *pshm* bit is set, the application has pinned shared memory segments, which may need to be detached. The memory remove request can succeed in any case, if there is enough pinnable memory in the system, so an action in this case is not necessarily required. The field *sys_pinnable_frames* provides this information, however, this value and other statistical values are just approximations. They reflect the state of the system at the time of the request. They are not updated during the request. The current size of physical memory can be determined by referencing the *_system_configuration.physmem* field.

To provide support for virtual real memory related DR operations, a new field, *dr_op*, has been added to the **dr_info** structure. The *dr_op* field provides information about the current DR operation. Additionally, all future DR operations use this field and the previously used resource bits will no longer be extended.

dr_wlm_info Structure

```
typedef struct dr_wlm_info {
  unsigned int cpu_add : 1; // cpu wlm resource add for the WPAR
  unsigned int cpu_rem : 1; // cpu wlm resource remove for the WPAR
  unsigned int mem_add : 1; // memory wlm resource add for the WPAR
  unsigned int mem_rem : 1; // memory wlm resource remove for the WPAR
  unsigned int rs_cpu  : 1; // wlm cpu rset change for the WPAR
  unsigned int rs_mem  : 1; // wlm memory rset change for the WPAR
  unsigned int pad1    : 2; // un-used
  unsigned int cpu_cap : 8; // percentage of cpu capacity of the WPAR
  unsigned int mem_cap : 8; // percentage of the memory capacity of the WPAR
  unsigned int pad2    : 8; // un-used
} dr_wlm_info_t;
```

dr_info Structure

```
typedef struct dr_info {
  unsigned int add : 1; // add request
               rem : 1; // remove request
               cpu : 1; // target resource is a cpu
               mem : 1; // target resource is memory
             check : 1; // check phase in effect
               pre : 1; // pre phase in effect
              post : 1; // post phase in effect
         posterror : 1; // post error phase in effect
```

```
             force : 1; // force option is in effect
          bindproc : 1; // process has bindprocessor dependency
          softpset : 1; // process has WLM software partition dependency
          hardpset : 1; // process has processor set API dependency
             plock : 1; // process has plock'd memory
              pshm : 1; // process has pinned shared memory
           ent_cap : 1; // target resource:entitled capacity
           var_wgt : 1; // target resource:variable weight
    splpar_capable : 1; // 1/0 partition is/not splpar capable
     splpar_shared : 1; // 1/0 partition shared/dedicated mode
     splpar_capped : 1; // 1/0 partition capped/uncapped mode
       splpar_constrained : 1;  // Set to 1 if requested capacity
                            update is constrained by PHYP to
                            be within partition capacity bounds.
                         //

    unsigned int migrate : 1;    // migration operation
    unsigned int hibernate : 1;    // hibernation operation
    unsigned int partition : 1;    // resource is partition
    unsigned int topology_update : 1; // topology update

    // The following fields are filled out for cpu based requests
    int lcpu;   // logical cpu ID being added or removed
    int bcpu;   // bind cpu ID being added or removed

    // The following fields are filled out for memory based requests
    size64_t  req_memsz_change;     // User request size in bytes
    size64_t  sys_memsz;            // System Memory size at time of request
    rpn64_t   sys_free_frames;      // Number of free frames in the system
    rpn64_t   sys_pinnable_frames;  // Number of pinnable frames in system
    rpn64_t   sys_total_frames;     // Total number of frames in system


    // SPLPAR parameters.
    uint64_t  capacity;   // partition current entitled capacity
                                           if ent_cap bit is set, partition's
                                           current variable capacity weight
                                           if var_wgt bit is set.
                                        //

    int   delta_cap;      // delta capacity added/removed to
                                           current value depending on add/rem
                                           bit flag value above
                                        //
    dr_wlm_info_t   dr_wlm;     // DR info for the WPAR
    ushort     dr_op;  // type of DR operation

    ushort     dr_pad;     // reserved pad field
    size64_t     mem_capacity; // partition's entitled
I/O memory or variable capacity.
    ssize64_t   delta_mem_capacity; // amount of I/O being added/removed

    int  reserved[2];

} dr_info_t;
```

Parameters

Item	Description
flags	The following values are supported: **DR_QUERY** Identifies the current DLPAR request and the actions that the application must take to comply with the current DLPAR request. This information is returned to the caller in the structure identified by the *dr_info* parameter. **DR_EVENT_FAIL** Fail the current DLPAR event. Root authority is required. **DR_RECONFIG_DONE** This flag is used with the **DR_QUERY** flag. The application notifies the kernel that the actions it took to comply with the current DLPAR request are now complete. The **dr_info** structure identifying the DLPAR request that was returned is passed as an input parameter.
dr_info	Contains the address of a **dr_info** structure, which is declared with the address space of the application.

Return Values

Upon success, the **dr_reconfig** system call returns a zero. If unsuccessful, it returns negative one and sets the **errno** variable to the appropriate error value.

Error Codes

Item	Description
EINVAL	Invalid flags.
ENXIO	No DLPAR event in progress.
EPERM	Root authority required for DR_EVENT_FAIL.
EINPROGRESS	Cancellation of DLPAR event may only occur in the check phase.

Related information:

Making Programs DLPAR-Aware Using DLPAR APIs

e

The following kernel services begin with the with the letter e.

e_assert_wait Kernel Service

Purpose

Asserts that the calling kernel thread is going to sleep.

Syntax

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

void e_assert_wait ( event_word,  interruptible)
tid_t *event_word;
boolean_t interruptible;
```

Parameters

Item	Description
event_word	Specifies the shared event word. The kernel uses the *event_word* parameter as the anchor to the list of threads waiting on this shared event.
interruptible	Specifies if the sleep is interruptible.

Description

The **e_assert_wait** kernel service asserts that the calling kernel thread is about to be placed on the event list anchored by the *event_word* parameter. The *interruptible* parameter indicates wether the sleep can be interrupted.

This kernel service gives the caller the opportunity to release multiple locks and sleep atomically without losing the event should it occur. This call is typically followed by a call to either the **e_clear_wait** or **e_block_thread** kernel service. If only a single lock needs to be released, then the **e_sleep_thread** kernel service should be used instead.

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

Execution Environment

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

Related reference:

"e_clear_wait Kernel Service" on page 113

"e_sleep_thread Kernel Service" on page 116

Related information:

Process and Exception Management Kernel Services

e_block_thread Kernel Service

Purpose

Blocks the calling kernel thread.

Syntax

```
#include <sys/sleep.h>
int e_block_thread ()
```

Description

The **e_block_thread** kernel service blocks the calling kernel thread. The thread must have issued a request to sleep (by calling the **e_assert_wait** kernel service). If it has been removed from its event list, it remains runnable.

Execution Environment

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

Return Values

The **e_block_thread** kernel service return a value that indicate how the thread was awakened. The following values are defined:

Item	Description
THREAD_AWAKENED	Denotes a normal wakeup; the event occurred.
THREAD_INTERRUPTED	Denotes an interruption by a signal.
THREAD_TIMED_OUT	Denotes a timeout expiration.
THREAD_OTHER	Delineates the predefined system codes from those that need to be defined at the subsystem level. Subsystem should define their own values greater than or equal to this value.

Related reference:

"e_assert_wait Kernel Service" on page 111

Related information:

Process and Exception Management Kernel Services

e_clear_wait Kernel Service

Purpose

Clears the wait condition for a kernel thread.

Syntax

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

void e_clear_wait ( tid,  result)
tid_t tid;
int result;
```

Parameters

Item **Description**

tid Specifies the kernel thread to be awakened.

result Specifies the value returned to the awakened kernel thread. The following values can be used:

THREAD_AWAKENED
Usually generated by the **e_wakeup** or **e_wakeup_one** kernel service to indicate a normal wakeup.

THREAD_INTERRUPTED
Indicates an interrupted sleep. This value is usually generated by a signal delivery when the **INTERRUPTIBLE** flag is set.

THREAD_TIMED_OUT
Indicates a timeout expiration.

THREAD_OTHER
Delineates the predefined system codes from those that need to be defined at the subsystem level. Subsystem should define their own values greater than or equal to this value.

Description

The **e_clear_wait** kernel service clears the wait condition for the kernel thread specified by the *tid* parameter, and the thread is made runnable.

This kernel service differs from the **e_wakeup**, **e_wakeup_one**, and **e_wakeup_w_result** kernel services in the fact that it assumes the identity of the thread to be awakened. This kernel service should be used to handle exceptional cases, where a special action needs to be taken. The *result* parameter is used to specify the value returned to the awakened thread by the **e_block_thread** or **e_sleep_thread** kernel service.

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

Execution Environment

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

Related reference:

“e_wakeup, e_wakeup_one, or e_wakeup_w_result Kernel Service” on page 120

“e_block_thread Kernel Service” on page 112

Related information:

Process and Exception Management Kernel Services

e_sleep Kernel Service

Purpose

Forces the calling kernel thread to wait for the occurrence of a shared event.

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/sleep.h> int e_sleep (*event_word*, *flags***)** **tid_t** **event_word*; **int** *flags*;

Parameters

Item	Description
event_word	Specifies the shared event word. The kernel uses the *event_word* parameter to anchor the list of processes sleeping on this event. The *event_word* parameter must be initialized to **EVENT_NULL** before its first use.
flags	Specifies the flags that control action on occurrence of signals. These flags can be found in the **/usr/include/sys/sleep.h** file. The *flags* parameter is used to control how signals affect waiting for an event. The following flags are available to the **e_sleep** service: **EVENT_SIGRET** Indicates the termination of the wait for the event by an unmasked signal. The return value is set to **EVENT_SIG**. **EVENT_SIGWAKE** Indicates the termination of the event by an unmasked signal. This flag results in the transfer of control to the return from the last **setjmpx** service with the return value set to **EINTR**. **EVENT_SHORT** Prohibits the wait from being terminated by a signal. This flag should only be used for short, guaranteed-to-wakeup sleeps.

Description

The **e_sleep** kernel service is used to wait for the specified shared event to occur. The kernel places the current kernel thread on the list anchored by the *event_word* parameter. This list is used by the **e_wakeup** service to wake up all threads waiting for the event to occur.

The anchor for the event list, the *event_word* parameter, must be initialized to **EVENT_NULL** before its first use. Kernel extensions must not alter this anchor while it is in use.

The **e_wakeup** service does not wake up a thread that is not currently sleeping in the **e_sleep** function. That is, if an **e_wakeup** operation for an event is issued before the process calls the **e_sleep** service for the event, the thread still sleeps, waiting on the next **e_wakeup** service for the event. This implies that routines using this capability must ensure that no timing window exists in which events could be missed due to the **e_wakeup** service being called before the **e_sleep** operation for the event has been called.

Note: The **e_sleep** service can be called with interrupts disabled only if the event or lock word is pinned.

Execution Environment

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

Return Values

Item	Description
EVENT_SUCC	Indicates a successful operation.
EVENT_SIG	Indicates that the **EVENT_SIGRET** flag is set and the wait is terminated by a signal.

Related reference:

"e_sleepl Kernel Service"

"e_wakeup, e_wakeup_one, or e_wakeup_w_result Kernel Service" on page 120

Related information:

Process and Exception Management Kernel Services

e_sleepl Kernel Service

Purpose

Forces the calling kernel thread to wait for the occurrence of a shared event.

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/sleep.h> int e_sleepl (*lock_word***,** *event_word***,** *flags***) int** **lock_word***; tid_t** **event_word***; int** *flags***;**

Parameters

Item	Description
lock_word	Specifies the lock word for a conventional process lock.
event_word	Specifies the shared event word. The kernel uses this word to anchor the list of kernel threads sleeping on this event. This event word must be initialized to **EVENT_NULL** before its first use.
flags	Specifies the flags that control action on occurrence of a signal. These flags are found in the **/usr/include/sys/sleep.h** file.

Description

Note: The **e_sleepl** kernel service is provided for porting old applications written for previous versions of the operating system. Use the **e_sleep_thread** kernel service when writing new applications.

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

The **e_wakeup** service does not wake up a thread that is not currently sleeping in the **e_sleepl** function. That is, if an **e_wakeup** operation for an event is issued before the thread calls the **e_sleepl** service for the event, the thread still sleeps, waiting on the next **e_wakeup** operation for the event. This implies that routines using this capability must ensure that no timing window exists in which events could be missed due to the **e_wakeup** service being called before the **e_sleepl** service for the event has been called.

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

The anchor for the event list, specified by the *event_word* parameter, must be initialized to **EVENT_NULL** before its first use. Kernel extensions must not alter this anchor while it is in use.

Note: The **e_sleepl** service can be called with interrupts disabled, only if the event or lock word is pinned.

Values for the flags Parameter

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

Item	Description
EVENT_SIGRET	Indicates the termination of the wait for the event by an unmasked signal. The return value is set to EVENT_SIG.
EVENT_SIGWAKE	Indicates the termination of the event by an unmasked signal. This flag also indicates the transfer of control to the return from the last **setjmpx** service with the return value set to **EINTR**.
EVENT_SHORT	Indicates that signals cannot terminate the wait. Use the **EVENT_SHORT** flag for only short, guaranteed-to-wakeup sleeps.

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

Execution Environment

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

Return Values

Item	Description
EVENT_SUCC	Indicates successful completion.
EVENT_SIG	Indicates that the **EVENT_SIGRET** flag is set and the wait is terminated by a signal.

Related reference:
"e_sleep Kernel Service" on page 114
"e_wakeup, e_wakeup_one, or e_wakeup_w_result Kernel Service" on page 120
Related information:
Interrupt Environment

e_sleep_thread Kernel Service

Purpose

Forces the calling kernel thread to wait for the occurrence of a shared event.

Syntax

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

int e_sleep_thread ( event_word,  lock_word,  flags)
tid_t *event_word;
void *lock_word;
int flags;
```

Parameters

Item	Description
event_word	Specifies the shared event word. The kernel uses the *event_word* parameter as the anchor to the list of threads waiting on this shared event.
lock_word	Specifies simple or complex lock to unlock.
flags	Specifies lock and signal handling options.

Description

The **e_sleep_thread** kernel service forces the calling thread to wait until a shared event occurs. The kernel places the calling thread on the event list anchored by the *event_word* parameter. This list is used by the **e_wakeup**, **e_wakeup_one**, and **e_wakeup_w_result** kernel services to wakeup some or all threads waiting for the event to occur.

A lock can be specified; it will be unlocked when the kernel service is entered, just before the thread blocks. This lock can be a simple or a complex lock, as specified by the *flags* parameter. When the kernel service exits, the lock is re-acquired.

Flags

The *flags* parameter specifies options for the kernel service. Several flags can be combined with the bitwise OR operator. They are described below.

The four following flags specify the lock type. If the *lock_word* parameter is not **NULL**, exactly one of these flags must be used.

Flag	Description
LOCK_HANDLER	*lock_word* specifies a simple lock protecting a thread-interrupt or interrupt-interrupt critical section.
LOCK_SIMPLE	*lock_word* specifies a simple lock protecting a thread-thread critical section.
LOCK_READ	*lock_word* specifies a complex lock in shared-read mode.
LOCK_WRITE	*lock_word* specifies a complex lock in exclusive write mode.

The following flag specify the signal handling. By default, while the thread sleeps, signals are held pending until it wakes up.

Item	Description
INTERRUPTIBLE	The signals must be checked while the kernel thread is sleeping. If a signal needs to be delivered, the thread is awakened.

Return Values

The **e_sleep_thread** kernel service return a value that indicate how the kernel thread was awakened. The following values are defined:

Item	Description
THREAD_AWAKENED	Denotes a normal wakeup; the event occurred.
THREAD_INTERRUPTED	Denotes an interruption by a signal. This value can be returned even if the **INTERRUPTIBLE** flag is not set since it may be also generated by the **e_clear_wait** or **e_wakeup_w_result** kernel services.
THREAD_TIMED_OUT	Denotes a timeout expiration. The **e_sleep_thread** has no timeout. However, the **e_clear_wait** or **e_wakeup_w_result** kernel services may generate this return value.
THREAD_OTHER	Delineates the predefined system codes from those that need to be defined at the subsystem level. Subsystem should define their own values greater than or equal to this value.

Execution Environment

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

Related reference:

"e_wakeup, e_wakeup_one, or e_wakeup_w_result Kernel Service" on page 120

"e_block_thread Kernel Service" on page 112

Related information:

Locking Kernel Services

et_post Kernel Service

Purpose

Notifies a kernel thread of the occurrence of one or more events.

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/sleep.h> void et_post (*events*, *tid***)**
unsigned long *events***; tid_t** *tid***;**

Parameters

Item	**Description**
events	Identifies the masks of events to be posted.
tid	Specifies the thread identifier of the kernel thread to be notified.

Description

The **et_post** kernel service is used to notify a kernel thread that one or more events occurred.

The **et_post** service provides the fastest method of interprocess communication, although only the event numbers are passed.

The event numbers must be known by the cooperating components, either through programming convention or the passing of initialization parameters.

The **et_post** service is performed automatically when sending a request to a device queue serviced by a kernel thread or when sending an acknowledgment.

The **EVENT_KERNEL** mask defines the event bits reserved for use by the kernel. For example, a bit with a value of 1 indicates an event bit reserved for the kernel. Kernel extensions should assign their events starting with the most significant bits and working down. If threads using the **et_post** service are also using the device queue management kernel extensions, care must be taken not to use the event bits registered for device queue management.

The **et_wait** service does not sleep but returns immediately if a specified event has already been posted by the **et_post** service.

Execution Environment

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

Return Values

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

Related reference:

"et_wait Kernel Service"

Related information:

Process and Exception Management Kernel Services

et_wait Kernel Service

Purpose

Forces the calling kernel thread to wait for the occurrence of an event.

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/sleep.h> unsigned long et_wait (*wait_mask*, *clear_mask*, *flags*) **unsigned long** *wait_mask*; **unsigned long** *clear_mask*; **int** *flags*;

Parameters

Item	Description
wait_mask	Specifies the mask of events to await.
clear_mask	Specifies the mask of events to clear.
flags	Specifies the flags controling actions on occurrence of a signal.

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

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

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

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

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

Description

The **et_wait** kernel service forces the calling kernel thread to wait for specified events to occur.

The *wait_mask* parameter indicates a mask, where each bit set equal to 1 represents an event for which the thread must wait. The *clear_mask* parameter indicates a mask of events that must clear when the wait is complete. Subsequent calls to the **et_wait** service return immediately unless you clear the bits, which ends the wait.

Note: The **et_wait** service can be called with interrupts disabled only if the event or lock word is pinned.

Strategies for Using et_wait

Calling the **et_wait** kernel service with the **EVENT_SIGRET** flag clears the the pending events field when the signal is received. If **et_wait** is called again by the same kernel thread, the thread waits indefinitely for an event that has already occurred. When this happens, the thread does not run to completion. This problem occurs only if the event and signal are posted at the same time.

To avoid this problem, use one of the following programming methods:

- Use the **EVENT_SHORT** flag to prevent signals from waking the thread up.
- Mask signals prior to the call of **et_wait** by using the limit_sigs kernel service. Then call **et_wait**. Invoke the sigprocmask call to restore the signal mask by using the mask returned previously by limit_sigs.

The **et_wait** service is also used to clear events without waiting for them to occur. This is accomplished by doing one of the following:

- Set the *wait_mask* parameter to **EVENT_NDELAY**.
- Set the bits in the *clear_mask* parameter that correspond with the events to be cleared to 1.

Because the **et_wait** service returns an event mask indicating those events that were actually cleared, these methods can be used to poll the events.

Execution Environment

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

Return Values

Upon successful completion, the **et_wait** service returns an event mask indicating the events that terminated the wait. If an **EVENT_NDELAY** value is specified, the returned event mask indicates the pending events that were cleared by this call. Otherwise, it returns the following error code:

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

Related reference:

"et_post Kernel Service" on page 118

"setjmpx Kernel Service" on page 467

Related information:

Process and Exception Management Kernel Services

e_wakeup, e_wakeup_one, or e_wakeup_w_result Kernel Service

Purpose

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

Syntax

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

void e_wakeup ( event_word)
tid_t *event_word;

void e_wakeup_one ( event_word)
tid_t *event_word;

void e_wakeup_w_result ( event_word,  result)
tid_t *event_word;
int result;
```

Parameters

Item	Description
event_word	Specifies the shared event designator. The kernel uses the *event_word* parameter as the anchor to the list of threads waiting on this shared event.
result	Specifies the value returned to the awakened kernel thread. The following values can be used:

THREAD_AWAKENED
Indicates a normal wakeup. This is the value automatically generated by the **e_wakeup** or **e_wakeup_one** kernel services.

THREAD_INTERRUPTED
Indicates an interrupted sleep. This value is usually generated by a signal delivery when the **INTERRUPTIBLE** flag is set.

THREAD_TIMED_OUT
Indicates a timeout expiration.

THREAD_OTHER
Delineates the predefined system codes from those that need to be defined at the subsystem level. Subsystem should define their own values greater than or equal to this value.

Description

The **e_wakeup** and **e_wakeup_w_result** kernel services wake up all kernel threads sleeping on the event list anchored by the *event_word* parameter. The **e_wakeup_one** kernel service wakes up only the most favored thread sleeping on the event list anchored by the *event_word* parameter.

When threads are awakened, they return from a call to either the **e_block_thread** or **e_sleep_thread** kernel service. The return value depends on the kernel service called to wake up the threads (the wake-up kernel service):

- **THREAD_AWAKENED** is returned if the **e_wakeup** or **e_wakeup_one** kernel service is called
- The value of the *result* parameter is returned if the **e_wakeup_w_result** kernel service is called.

If a signal is delivered to a thread being awakened by one of the wake-up kernel services, and if the thread specified the **INTERRUPTIBLE** flag, the signal delivery takes precedence. The thread is awakened with a return value of **THREAD_INTERRUPTED**, regardless of the called wake-up kernel service.

The **e_wakeup** and **e_wakeup_w_result** kernel services set the *event_word* parameter to **EVENT_NULL**.

The **e_wakeup**, **e_wakeup_one**, and **e_wakeup_w_result** kernel services have no return values.

Execution Environment

The **e_wakeup**, **e_wakeup_one**, and **e_wakeup_w_result** kernel services can be called from either the process environment or the interrupt environment.

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

Related reference:

"e_clear_wait Kernel Service" on page 113

"e_sleep_thread Kernel Service" on page 116

Related information:

Process and Exception Management Kernel Services

e_wakeup_w_sig Kernel Service

Purpose

Posts a signal to sleeping kernel threads.

Syntax

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

void e_wakeup_w_sig ( event_word,  sig)
tid_t *event_word;
int sig;
```

Parameters

Item	Description
event_word	Specifies the shared event word. The kernel uses the *event_word* parameter as the anchor to the list of threads waiting on this shared event.
sig	Specifies the signal number to post.

Description

The **e_wakeup_w_sig** kernel service posts the signal *sig* to each kernel thread sleeping interruptible on the event list anchored by the *event_word* parameter.

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

Execution Environment

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

Related reference:

"e_block_thread Kernel Service" on page 112

"e_clear_wait Kernel Service" on page 113

Related information:

Process and Exception Management Kernel Services

eeh_broadcast Kernel Service

Purpose

This service is provided for device drivers to coordinate activities during an EEH event.

Syntax

```
void eeh_broadcast(handle, message)
eeh_handle_t    handle;
unsigned long long message;
```

Parameters

Item	Description
handle	EEH handle obtained from **eeh_init** or **eeh_init_multifunc**
message	User- or kernel-defined message

Description

Because single-function drivers do not have a need for coordination, this service is intended for multifunction drivers only. If a single-function driver calls it, it is a NOP. There are two kinds of messages that can be sent among the drivers: kernel-defined messages (such as EEH_DD_SUSPEND and EEH_DD_DEAD) and the user-defined messages. See **sys/eeh.h** for help on how to define user messages.

Kernel messages have a higher priority than user messages. Therefore, if user messages and kernel messages are both pending, the kernel messages are sent out before the user messages.

Note: Device drivers should only broadcast their own messages (that is, the user-defined message) and not the kernel messages.

Within the kernel messages, EEH_DD_DEAD has the highest priority. Multiple messages of the same kind may or may not be coalesced depending upon the relative timing. Messages are sent by invoking the callback routines. The callback routines are invoked sequentially but not in any specific order except that the last driver to receive a message will have the EEH_MASTER flag set to indicate that all other drivers have finished processing the message. Only one message is broadcast at a time—that is, all registered callback routines are called sequentially with the same message before moving on to the next message. Finally, they are invoked asynchronously at INTIODONE priority. Because they are broadcast asynchronously, a device driver must not assume on a specific timeout within which the message would arrive.

The macro **EEH_BROADCAST**(*handle*, *message*) is provided for device drivers to call this service.

Execution Environment

This kernel service can be called from the process or interrupt environment.

Return Values

This service has no return value.

Related reference:

"eeh_enable_slot Kernel Service" on page 127

"eeh_init_multifunc Kernel Service" on page 129

"eeh_slot_error Kernel Service" on page 135

eeh_clear Kernel Service

Purpose

This service unregisters a slot for an EEH function and removes resources allocated by the **eeh_init** or **eeh_init_multifunc** kernel service.

Syntax

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

void eeh_clear(handle)
eeh_handle_t handle;
```

Parameters

Item	**Description**
handle	EEH handle obtained from the**eeh_init** or **eeh_init_multifunc** kernel services

Description

Single-function Drivers: This service disables EEH function on the slot and frees its **eeh_handle**.

Multifunction Drivers: For a multifunction adapter driver, this service removes the driver from a list of registered drivers under the same parent bus. This service also disables EEH function on the slot if this is the last driver to unregister and the state of the slot is NORMAL.

All device drivers are required to call **eeh_clear** before being removed from the system, so that there are no hot plug conflicts. A subsequent adapter might fail in **eeh_init_multifunc()** on the slot if the **eeh_clear** kernel service has not cleared the prior device drivers on that slot. A driver can unregister at unconfigure/unload time. The kernel checks the state of the slot when this service is called. If the slot state is neither NORMAL nor DEAD, **eeh_clear** sleeps until the state returns to one of them.

The macro **EEH_CLEAR**(*handle*) is provided for device drivers to call this service. This service is called by a function pointer in the EEH handle.

Execution Environment

This kernel service can only be called from the process environment.

Return Values

This service has no return values.

Related reference:

"eeh_broadcast Kernel Service" on page 122

"eeh_enable_slot Kernel Service" on page 127

"eeh_init Kernel Service" on page 128

eeh_disable_slot Kernel Service

Purpose

This service disables a slot for the EEH operations.

Syntax

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

long eeh_disable_slot(handle)
eeh_handle_t handle;
```

Parameters

Item	Description
handle	EEH handle obtained from the**eeh_init** kernel service

Description

This service disables EEH operation on a slot.

CAUTION:
CAUTION: Disabling EEH operation on a slot is highly discouraged, because it can cause system crash or worse, data corruption.

This service can only be called by the single-function adapter drivers. If the service fails for a hardware or firmware reason, an error is logged.

Multifunction drivers call this service indirectly via **eeh_clear()**. It fails with EEH_FAIL if called directly by a multifunction driver.

The macro **EEH_DISABLE_SLOT**(*handle*) is provided for device drivers to call this service.

Execution Environment

This kernel service can be called from the process or interrupt environment.

Return Values

Item	Description
EEH_SUCC	Slot successfully disabled
EEH_FAIL	Unable to disable the slot

Related reference:
"eeh_enable_slot Kernel Service" on page 127
"eeh_read_slot_state Kernel Service" on page 132
"eeh_slot_error Kernel Service" on page 135

eeh_enable_dma Kernel Service

Purpose

This service enables DMA operations to an adapter after an EEH event.

Syntax

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

long eeh_enable_dma(handle)
eeh_handle_t handle;
```

Parameters

Item	Description
handle	EEH handle obtained from the**eeh_init** or **eeh_init_multifunc** kernel services

Description

When an EEH event occurs on a slot, all Direct Memory Access (DMA) operations on the slot are inhibited. This service should be called to re-enable DMA after an EEH event. This service can only be called from the dump context (that is, when the dump is in progress).

Single-function Drivers: This service enables the DMA operations on a slot. If this call fails with EEH_FAIL, an error is logged by the kernel.

Multifunction Drivers: On the multifunction adapters, the slot state must be either SUSPEND or DEBUG, and the caller must be an EEH_MASTER. This service is called only from a dump context. While a system dump is in progress, all callbacks and broadcasts are suspended, and a multifunction adapter is treated like a single-function adapter, because the system can no longer support the EEH multifunction kernel services. If the service fails, EEH_FAIL is returned. If the failure is due to hardware or firmware, an error is logged.

There are cases when this kernel service cannot succeed because of the platform state restrictions. In such a case, if a driver calls it, the service would return EEH_FAIL. This causes the slot to be marked permanently unavailable, which is not correct because the slot can be recovered. To avoid receiving EEH_FAIL from this service, the driver should supply the EEH_ENABLE_NO_SUPPORT_RC flag at **eeh_init_multifunc()** time. If the EEH_ENABLE_NO_SUPPORT_RC flag is supplied, **eeh_enable_dma()** returns EEH_NO_SUPPORT, indicating to the drivers that they cannot collect debug data but must continue with the next step in recovery.

The macro **EEH_ENABLE_DMA**(*handle*) is provided for device drivers to call this service.

Execution Environment

This kernel service can only be called from a process or interrupt environment.

Return Values

This kernel service has no return values.

Related reference:

"eeh_disable_slot Kernel Service" on page 124

"eeh_enable_pio Kernel Service"

"eeh_enable_slot Kernel Service" on page 127

eeh_enable_pio Kernel Service

Purpose

This kernel service enables programmed I/O (PIO or MMIO) to an adapter after an EEH event.

Syntax

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

long eeh_enable_pio(handle)
eeh_handle_t handle;
```

Parameters

Item	Description
handle	EEH handle obtained from the **eeh_init** or **eeh_init_multifunc** kernel services

Description

When an EEH event occurs on a slot, all load and store operations (such as PIO) are inhibited. This kernel service should be called to re-enable PIO after an EEH event.

Single-function Drivers: This kernel service enables the load and store operations on a slot. If this call fails with EEH_FAIL, an error is logged by the kernel.

Multifunction Drivers: On the multifunction adapters, the state of the slot is checked for either SUSPEND or DEBUG. The caller must be an EEH_MASTER. If the state is SUSPEND, a series of device driver callback routines is executed with a command option of EEH_DD_DEBUG and flag set to EEH_DD_PIO_ENABLED. The callbacks inform device drivers that PIO has been enabled and that further debug procedures can be executed (such as reading command and status register). This service can be called as a result of the EEH_DD_SUSPEND or EEH_DD_DEBUG callback message as many times as needed by the EEH_MASTER. Additional calls to this service trigger a new set of callbacks. If this service fails, EEH_FAIL is returned. If the failure is due to hardware or firmware, an error is logged.

There are cases when this kernel service cannot succeed due to the platform state restrictions. In such a case, if a driver calls it, the kernel service would return EEH_FAIL followed by a EEH_DD_DEAD message. This causes the slot to be marked permanently unavailable, which is not correct because the slot can be recovered. To avoid receiving EEH_FAIL from this service, the driver should supply the EEH_ENABLE_NO_SUPPORT_RC flag at **eeh_init_multifunc()** time. If the EEH_ENABLE_NO_SUPPORT_RC flag is supplied, **eeh_enable_pio()** returns EEH_NO_SUPPORT, indicating to the drivers that they cannot collect debug data but must continue with the next step in recovery.

The macro **EEH_CLEAR**(*handle*) is provided for device drivers to call this service. This service is called via a function pointer in the EEH handle.

Note: Enabling PIO is not the same as recovering the slot. In fact, this is an optional step in the recovery procedure.

Execution Environment

This kernel service can be called from the process or interrupt environment.

Return Values

Item	Description
EEH_SUCC	PIO successfully enabled.
EEH_FAIL	Invalid call or could not enable PIO.
EEH_NO_SUPPORT	Call is valid according to AIX EEH state, but current platform state precludes normal completion.

Related reference:

"eeh_disable_slot Kernel Service" on page 124

"eeh_enable_dma Kernel Service" on page 125

"eeh_enable_slot Kernel Service"

eeh_enable_slot Kernel Service

Purpose

This service enables a slot for the EEH operations.

Syntax

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

long eeh_enable_slot(handle)
eeh_handle_t handle;
```

Parameters

Item	Description
handle	EEH handle obtained from the**eeh_init** kernel service

Description

This service enables EEH operation on a slot so that when certain errors occur on a PCI bus, the slot will freeze (that is, PIO and DMA are disabled, which prevents potential system crash, data corruption, and so on). This service can only be called by the single-function adapter drivers. If the service fails for hardware or firmware reasons, an error is logged.

Multifunction drivers call this service indirectly via **eeh_init_multifunc()**. It fails with EEH_FAIL if called directly by a multifunction driver.

The macro **EEH_ENABLE_SLOT**(*handle*) is provided for device drivers to call this service.

Execution Environment

This kernel service can be called from the process or interrupt environment.

Return Values

Item	Description
EEH_SUCC	Slot successfully enabled
EEH_FAIL	Unable to enable the slot

Related reference:

"eeh_disable_slot Kernel Service" on page 124

"eeh_enable_dma Kernel Service" on page 125

"eeh_enable_pio Kernel Service" on page 126

eeh_init Kernel Service

Purpose

This service registers a single-function adapter slot on a PCI/PCI-E bus for EEH function.

Syntax

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

eeh_handle_t eeh_init(pbid, slot, flag)
long     pbid;
long     slot;
long     flag;
```

Parameters

Item	Description
pbid	AIX parent bus identifier
slot	device slot (device*8+function). This is same as "connwhere" property in CuDv.
flag	flag that enables eeh

Description

The *pbid* argument identifies a bus type and number. The bus type is IO_PCI in the case of PCI and PCI-X bus. If the bus type is IO_PCIE, the device is on PCI-E (PCI Express) bus. The bus number is a unique identifier determined during bus configuration. The **BID_VAL** macro defined in **ioacc.h** is used to generate the *bid*. The *slot* argument is the device/function combination ((device*8) + function) as in the PCI addressing scheme. The *flag* argument of EEH_ENABLE enables the slot. The *flag* argument of EEH_DISABLE does not enable the slot but still allocates an EEH handle. This service should be called only by the single-function adapter drivers.

The macro **EEH_INIT**(*pbid*, *slot*, *flag*) is provided for the device drivers to call this service. The **eeh_handle** is defined as follows in <sys/eeh.h>:

```
/*
 * This is the eeh_handle structure for the eeh_* services
 */
typedef struct eeh_handle *       eeh_handle_t;
struct eeh_handle {
        struct  eeh_handle *next;
        long    bid;                    /* bus id passed to eeh_init    */
        long    slot;                   /* slot passed to eeh_init      */
        long    flag;                   /* flag passed to eeh_init      */
        int     config_addr;            /* Configuration Space Address  */
        int     eeh_mode;               /* Indicates safe mode          */
        uint    retry_delay;            /* re-read the slot state after *
                                         * these many seconds.          */
        int     reserved1;
        int     reserved2;
        int     reserved3;
        long long       PHB_Unit_ID;    /* /pci@               */
        void    (*eeh_clear)(eeh_handle_t);
        long    (*eeh_enable_pio)(eeh_handle_t);
        long    (*eeh_enable_dma)(eeh_handle_t);
        long    (*eeh_reset_slot)(eeh_handle_t, int);
        long    (*eeh_enable_slot)(eeh_handle_t);
        long    (*eeh_disable_slot)(eeh_handle_t);
        long    (*eeh_read_slot_state)(eeh_handle_t, long *, long *);
        long    (*eeh_slot_error)(eeh_handle_t, int, char *, long);
        struct eeh_shared_domain *parent_sd;    /* point back to the parent
                                         * shared domain structure if
                                         * in shared domain, NULL if singlefunc.
                                         */
        void    (*eeh_configure_bridge)(eeh_handle_t);
        void    (*eeh_broadcast)(eeh_handle_t, unsigned long long);
};
```

This is an exported kernel service.

Execution Environment

This service can only be called from the process environment.

Return Values

Item	Description
EEH_FAIL	Unable to allocate EEH handle.
EEH_NO_SUPPORT	EEH not supported on this system, no handle allocated.
struct eeh_handle *	If successful.

Related reference:

"eeh_broadcast Kernel Service" on page 122

"eeh_clear Kernel Service" on page 123

"eeh_enable_slot Kernel Service" on page 127

eeh_init_multifunc Kernel Service

Purpose

This kernel service registers a multifunction adapter slot on a PCI/PCI-E bus for EEH function.

Syntax

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

eeh_handle_t eeh_init_multifunc(gpbid, pbid, slot, flag, delay_seconds,
                                callback_ptr, dds_ptr)
long gpbid;
long pbid;
long slot;
long flag;
long delay_seconds;
long (*callback_ptr)();
void *dds_ptr;
```

Parameters

Item	Description
gpbid	Bus identifier of grandparent bus.
pbid	Bus identifier of parent bus.
slot	Slot on the parent bus (device*8+function). This is same as "connwhere" property in CuDv for the device.
flag	Flag that enables eeh, checks if the slot is already taken, etc.
delay_seconds	Time delay after a reset (in seconds).
callback_ptr	Device driver callback routine.
dds_ptr	Cookie to a target device driver that is usually a pointer to the adapter structure.

Description

This kernel service is provided for systems that support shared EEH domain, where one or more PCI functions in one or more adapters could belong to the same EEH recovery domain. In the past, this was called "multifunction adapter". The shared EEH domain is a more general concept than just a multifunction adapter. It is also recommended that single function adapters use the shared EEH model. All PCI-E devices, single or multifunction have to use the shared EEH model and hence this kernel service to register for EEH (instead of **eeh_init()**). In a shared EEH domain, multiple instances of device drivers may be operating. The instances are independent of each other and hence oblivious to each other's existence. Therefore, when recovering a slot from an EEH event, there is a need to coordinate the recovery procedure among them. As with **eeh_init()**, this service also returns an **eeh_handle** to the calling device driver.

There are two kinds of adapters: bridged and non-bridged. A bridged adapter has a bridge on the card such as PCI-to-PCI or PCIX-to-PCIX or PCI-E switch. For PCI and PCI-X bridged-adapters, *pbid* is the bus ID of the parent bus, and *gpbid* is the bus ID of the grandparent bus. The parent bus for a bridged adapter is the bus generated by the bridge/switch on the adapter. A *bid* identifies a bus number and type. The bus type is IO_PCI in the case of PCI and PCI-X bus, and IO_PCIE in the case of PCI-E bus. The bus number is a unique identifier determined during bus configuration. The **BID_VAL** macro defined in **ioacc.h** is used to generate the *bid*. For non-bridged adapters, *pbid* and *gpbid* are the same and are the bus IDs of the parent bus. Thus, when *pbid* and *gpbid* have different values for a PCI or PCI-X device, the kernel knows that this is a bridged adapter and needs to the bridge recovered as part of EEH recovery. It is not necessary to know if a PCI-E device is bridged or not for the purposes of EEH. Therefore, *pbid* and *gpbid* must be same and equal to the parent bus bid.

In summary, there are the following cases:

1. PCI/PCI-X non-bridged adapters and all PCI-E adapters: *gpbid* and *pbid* are same and equal to the parent bus *bid*.
2. PCI/PCI-X bridged adapters, *gpbid* is grandparent bus *bid*, and *pbid* is parent bus bid.

The *slot* argument is the device/function combination ((device* 8) + function) as in the PCI addressing scheme. This is the same as the **connwhere** ODM value of the device.

The following flag values are legal:

Item	Description
EEH_ENABLE_FLAG/EEH_DISABLE_FLAG	The slot is always enabled for EEH when this service is called by the first driver on that slot. All subsequent requests to enable the slot via the **EEH_ENABLE** flag are ignored. Therefore, the flag argument of **EEH_ENABLE** is optional, and a flag of **EEH_DISABLE** is ignored.
EEH_CHECK_SLOT	The flag argument of **EEH_CHECK_SLOT** verifies whether a given slot is already registered. A value of either **EEH_SLOT_ACTIVE** or **EEH_SLOT_FREE** is returned. No registration occurs with the **EEH_CHECK_SLOT** flag, and it supersedes all other flags. This flag simply checks the slot and returns without any other action.

Item	Description
EEH_ENABLE_NO_SUPPORT_RC	If the flag is set to **EEH_ENABLE_NO_SUPPORT_RC**, **eeh_enable_pio()** and **eeh_enable_dma()** return **EEH_NO_SUPPORT** under certain conditions. See "eeh_enable_dma Kernel Service" on page 125 and "eeh_enable_pio Kernel Service" on page 126 for more information.

Multiple flags can be logically ORed together.

The slot is always enabled for EEH when this service is called by the first driver on that slot. All subsequent requests to enable the slot via the EEH_ENABLE flag are ignored. Therefore, the *flag* argument of EEH_ENABLE is optional, and a flag of EEH_DISABLE is ignored. The flag argument of EEH_CHECK_SLOT verifies whether a given slot is already registered. A value of either EEH_SLOT_ACTIVE or EEH_SLOT_FREE is returned. No registration will occur with the EEH_CHECK_SLOT flag, and it supersedes all other flags. This flag just checks the slot and returns without any other action. If the flag is set to EEH_ENABLE_NO_SUPPORT_RC, **eeh_enable_pio()** and **eeh_enable_dma()** returns EEH_NO_SUPPORT under certain conditions. See **eeh_enable_pio()** and **eeh_enable_dma()** for more information. It is allowed to logically OR multiple flags together.

The *delay_seconds* argument allows the device driver to set a time delay between completion of PCI reset and configuration of the bridge on the adapter. The delay is enforced even if the adapter is non-bridged. If a value of 0 is specified for *delay_seconds*, a default delay time of 1 second is set. When several drivers register on the same *pbid* (under a shared EEH domain), the highest delay time among all registered drivers is used.

The *callback_ptr* argument is a function pointer to an EEH callback routine. The handler is defined by the device driver and is called by the kernel in order to coordinate recovery among different drivers on the same slot. The driver handles a variety of messages from the kernel in its callback routine. These messages trigger the next step in recovery. The callback routines are called sequentially at INTIODONE interrupt level.

The *dds_ptr* argument is a cookie that is passed to the driver when the callback routine is invoked. Drivers normally specify a pointer to the device driver's adapter structure.

EEH_SAFE mode: A bridged adapter needs to have its bridge reconfigured at the end of PCI reset. However, if the platform firmware does not support reconfiguration of the bridge, the adapter is marked as EEH_SAFE by the kernel. An EEH_SAFE adapter cannot finish error recovery after an EEH event because of the unsatisfied firmware dependency. See **eeh_reset_slot** for information on how the error recovery is handled in EEH_SAFE mode.

The macro **EEH_INIT_MULTIFUNC**(*gpbid*, *pbid*, *slot*, *flag*, *delay_seconds*, *callback_ptr*, *dds_ptr*) is provided for the device drivers in order to call this service. This is an exported kernel service.

Execution Environment

This kernel service can only be called from the process environment.

Return Values

Item	Description
EEH_FAIL	Unable to allocate EEH handle.
EEH_NO_SUPPORT	EEH is not supported on this system, no handle allocated.
EEH_SLOT_ACTIVE	Given slot is already registered.
EEH_SLOT_FREE	Given slot free.
EEH_BUSY	Unable to continue, because the slot is in the middle of error recovery.
struct eeh_handle *	Upon Success.

Related reference:

"eeh_broadcast Kernel Service" on page 122

"eeh_clear Kernel Service" on page 123

eeh_read_slot_state Kernel Service

Purpose

This service returns state and capabilities of a slot with respect to EEH operation.

Syntax

```
long eeh_read_slot_state(handle, state, support)
eeh_handle_t handle;
long *state;
long *support;
```

Parameters

Item	Description
handle	EEH handle obtained from **eeh_init** or **eeh_init_multifunc**
state	State of a slot with respect to EEH
support	Indicates if EEH is supported by this slot

Description

This service is used to query the hardware state of a slot and to determine whether a given slot supports EEH. It should be called to confirm an EEH event if the driver suspects that the PIO data is invalid (for example, getting all Fs from reading a register). This service returns the hardware state in *state* and indicates whether the slot supports EEH in *support*. The *state* and *support* parameters are integer values as shown below:

Valid *state* values are as follows:

Item	Description
EEH_NSTOPPED_RST_DEA	Reset deactivated and adapter is not in stopped state.
EEH_NSTOPPED_RST_ACT	Reset activated and adapter is not in stopped state.
EEH_STOPPED_LS_DIS	Adapter in stopped state with reset signal deactivated and Load/Store disabled.
EEH_STOPPED_LS_ENA	Adapter in stopped state with reset signal deactivated and Load/Store enabled.
EEH_UNAVAILABLE	Adapter is either permanently or temporarily unavailable.

Valid *support* values are as follows:

Item	Description
0	EEH not supported.
1	EEH supported.

The driver should call this service and check for EEH_STOPPED_LS_DIS and EEH_STOPPED_LS_ENA as the *state* values if it suspects an EEH event on the adapter. If the *state* is either of those values, the slot is said to be frozen.

Single-function Driver: A single-function adapter driver calls this service to query the state of the slot. If the service fails due to hardware or firmware reasons, an error is logged. If the service fails, *state* and *support* values are undefined, and EEH_FAIL is returned.

Multifunction Driver: For a multifunction adapter driver, this service analyzes the *state* to determine if:
- The state is frozen, or
- it is permanently unavailable (that is, the slot is unusable from hereon), or
- it is temporarily unavailable.

If the slot is in either a frozen or temporarily unavailable state, the EEH_DD_SUSPEND message is broadcast to all registered drivers on this slot. If the slot is permanently unavailable (that is, dead), the EEH_DD_DEAD message is broadcast. Upon receiving this message, the drivers are expected to suspend all further DMA, PIO, interrupt, configuration cycles, and so on until the slot is recovered. If the service fails due to hardware or firmware reasons, an error is logged, EEH_DD_DEAD is broadcast, and EEH_FAIL is returned.

Temporarily versus permanently unavailable state

In addition to *state* and *support*, this service also returns a valid *retry_delay* value in the **eeh_handle** structure if the *state* is EEH_UNAVAILABLE. If *retry_delay* is 0, it is permanently unavailable. If *retry_delay* is non-zero, it is temporarily unavailable. A permanently unavailable state means that the slot is unusable until a hot-plug operation or partition reboot is performed. Therefore, the drivers mark their adapters as unusable when they receive an EEH_UNAVAILABLE message (single-function) or when they receive an EEH_DD_DEAD message (multifunction). A temporarily unavailable state means that the current *state* of a slot is transient and might take a few minutes to settle down. Until that time, the device driver cannot begin recovery because it does not know what the final state will be. The temporarily unavailable state is handled differently by the single-function and multifunction drivers as follows:

Single-function Driver: Because a single-function driver drives its own recovery, it needs to check for *retry_delay* if the *state* is set to EEH_UNAVAILABLE. If *retry_delay* is non-zero, it represents the number of seconds that the driver should wait before calling this kernel service again. It continues to call this service repeatedly as long as the *state* is EEH_UNAVAILABLE and *retry_delay* is non-zero. Eventually, the *state* will end up in one of the following:
- EEH_NSTOPPED_RST_ACT
- EEH_STOPPED_LS_DIS
- EEH_UNAVAILABLE w/ "retry_delay" set to 0 (i.e. permanently unavailable)

At that point, the driver can continue with its normal course of action for a given state.

Multifunction Driver: A multifunction driver does not need to check for the *retry_delay* field when the state is EEH_UNAVAILABLE, because EEH_UNAVAILABLE would only mean permanently unavailable. In the case of temporarily unavailable, a multifunction driver would receive the EEH_DD_SUSPEND or EEH_DD_DEAD message after some time, depending upon the final *state* of the slot. If the final state was EEH_NSTOPPED_RST_ACT or EEH_STOPPED_LS_DIS, then EEH_DD_SUSPEND is broadcast; if it was EEH_UNAVAILABLE, then EEH_DD_DEAD is broadcast. Thus, from the point-of-view of a multifunction driver, there is no difference between frozen and temporarily unavailable.

The macro **EEH_READ_SLOT_STATE**(*handle*, *state*, *support*) is provided for device drivers to call this service.

Execution Environment

This kernel service can be called from the process or interrupt environment.

Return Values

Item	Description
EEH_SUCC	Successfully read the slot state and capabilities
EEH_FAIL	Unable to read the slot state and capabilities

Related reference:

"eeh_enable_slot Kernel Service" on page 127

"eeh_init Kernel Service" on page 128

"eeh_slot_error Kernel Service" on page 135

eeh_reset_slot Kernel Service

Purpose

This service activates, deactivates, or toggles the reset line of a PCI slot.

Syntax

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

long eeh_reset_slot(handle, flag)
eeh_handle_t handle;
long flag;
```

Parameters

Item	Description
handle	EEH handle obtained from the**eeh_init** or **eeh_init_multifunc** kernel services
flag	Flag can be either EEH_ACTIVE or EEH_DEACTIVE.

Description

Single-function Drivers: This service activates and deactivates the reset line between the Terminal Bridge and the adapter. The *flag* argument specifies whether to activate (EEH_ACTIVE) or deactivate (EEH_DEACTIVE) depending upon the required action. To do the reset of a slot, the reset line should be toggled by calling this service twice: once with EEH_ACTIVE followed by a second call with EEH_DEACTIVE. There should be a minimum of 100 milliseconds delay between the activation and deactivation of the signal. The minimum delay is specified by the PCI System Architecture and should be enforced by the single-function driver.

Multifunction Drivers: On a multifunction adapter, the EEH_MASTER for the slot drives error recovery. Therefore, only the EEH_MASTER can call this service. Unlike the single-function driver, the master calls this service only once with the EEH_ACTIVE flag.

For the multi-function drivers, the service first activates and then deactivates the reset signal on the slot. It enforces a 100–millisecond delay between the activation and deactivation as mandated by the PCI System Architecture. After the reset signal is deactivated, the service attempts to reconfigure the bridge on the adpater, if there is one (only applies to the bridged-adapters), after *dd_trb_timer* seconds specified in **eeh_init_multifunc()**. At the end of a successful reset and optional bridge recovery, an

EEH_DD_RESUME message is broadcast to the slot's multifunction drivers notifying them to resume normal operation. If this service fails, the EEH_DD_DEAD message is broadcast. If failure is due to hardware or firmware, an error is logged.

EEH_SAFE mode: If an EEH_SAFE adapter calls this service, the reset signal is activated but is never deactivated, thereby leaving the adapter in a "permanently unavailable" state. Such an adapter becomes available again if either the PCI hot-plug operation is performed on it or if the partition is rebooted. This service returns EEH_FAIL for an EEH_SAFE driver.

The macro **EEH_RESET_SLOT**(*handle*, *flag*) is provided for device drivers to call this service.

Execution Environment

This kernel service can be called from the process or interrupt environment.

Return Values

Item	Description
EEH_SUCC	Slot reset activate/deactivate succeeded
EEH_FAIL	Failed to activate/deactivate the reset line, nonmaster called the service, or EEH_SAFE mode is active
EEH_BUSY	Recovery is already in progress

Related reference:
“eeh_enable_slot Kernel Service” on page 127
“eeh_read_slot_state Kernel Service” on page 132
“eeh_slot_error Kernel Service”

eeh_slot_error Kernel Service

Purpose

This service logs a temporary or permanent error and optionally marks the slot permanently unavailable.

Syntax

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

long eeh_slot_error(handle, flag, dd_buf, dd_buf_length)
eeh_handle_t    handle;
int             flag;
char           *dd_buf;
long            dd_buf_length;
```

Parameters

Item	Description
handle	EEH handle obtained from **eeh_init** or **eeh_init_multifunc**
flag	EEH_RESET_TEMP or EEH_RESET_PERM
dd_buf	Address of the device driver's error log buffer
dd_buf_length	Length of device driver's error log buffer in bytes

Description

This service performs a number of tasks:

- It collects hardware data to help in understanding the nature and source of an EEH event
- It combines the device-driver-supplied debug data log with the hardware data log and creates an entry in the error log

- It optionally marks the slot permanently unavailable so that subsequent **eeh_read_slot_state()** calls return EEH_UNAVAILABLE with a *retry_delay* value of 0

The behavior of this kernel service is controlled by two *flag* values:

Item	Description
EEH_RESET_TEMP	This flag performs only the first two of the preceding tasks.
EEH_RESET_PERM	This flag performs all three tasks.

Depending on the hardware state of the slot, this service might not be able to collect the hardware data. Thus, the service succeeds but logs no data. If EEH_RESET_PERM was supplied, it still marks the slot permanently unavailable.

The *dd_buf* and *dd_buf_length* parameters are used to combine the device driver error log with the hardware log. The *dd_buf* argument is the address of an error log buffer that contains the device driver's data. The *dd_buf_length* argument is the length of this buffer. If the length exceeds *MAX_DD_LOG_SIZE* bytes, the driver's log data is truncated. If *dd_buf* is NULL, the error log contains only hardware data, if any.

Single-function driver: The kernel service works as in the preceding description. If it fails because of hardware or firmware reasons, EEH_FAIL is returned and an error is logged.

Multifunction driver: For the multifunction drivers, this service works as in the preceding description, except that if EEH_RESET_PERM was supplied, the EEH_DD_DEAD message is broadcast.

The macro **EEH_SLOT_ERROR**(*handle*, *flag*, *dd_buf*, *dd_buf_length*) is provided for device drivers to call this service.

Execution Environment

This kernel service can be called from the process or interrupt environment.

Return Values

Item	Description
EEH_SUCC	Successfully logged error
EEH_FAIL	Failed to log the error and optionally mark the slot permanently unavailable

Related reference:

"eeh_read_slot_state Kernel Service" on page 132

"eeh_reset_slot Kernel Service" on page 134

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

Item	Description
qe	Specifies the address of the request queue element.

Description

The **enque** kernel service is not part of the base kernel, but is provided by the device queue management kernel extension. This queue management kernel extension must be loaded into the kernel before loading any kernel extensions referencing these services.

The **enque** service places the queue element into a specified device queue. It is used for simple process-to-process communication within the kernel. The requester builds a copy of the queue element, indicated by the *qe* parameter, and passes this copy to the **enque** service. The kernel copies this queue element into a queue element in pinned global memory and then enqueues it on the target device queue.

The path identifier in the request queue element indicates the device queue into which the element is placed.

The **enque** service supports the sending of the following types of queue elements:

Queue Element	Description
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:

Operation Option	Description
ACK_COMPLETE	Acknowledge completion in all cases.
ACK_ERRORS	Acknowledge completion if the operation results in an error.
SYNC_REQUEST	Synchronous request.
CHAINED	Chained control blocks.
CONTROL_OPT	Kernel control operation.

Note: Only one of **ACK_COMPLETE**, **ACK_ERRORS**, or **SYNC_REQUEST** can be specified. Also, all of these options are ignored if the path specifies that no acknowledgment (**NO_ACK**) should be sent.

With the **SYNC_REQUEST** synchronous request option, control does not return from the **enque** service until the request queue element is acknowledged. This performs in one step what can also be achieved by sending a queue element with the **ACK_COMPLETE** flag on, and then calling either the **et_wait** or **waitq** kernel services.

The kernel calls the server's **check** routine, if one is defined, before a queue element is placed on the device queue. This routine can stop the operation if it detects an error.

The kernel notifies the device queue's server, if necessary, after a queue element is placed on the device queue. This is done by posting the server process (using the **et_post** kernel service) with an event control bit.

Execution Environment

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

Return Values

Item	Description
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.

Related reference:

"et_post Kernel Service" on page 118

"et_wait Kernel Service" on page 119

"waitq Kernel Service" on page 582

errresume Kernel Service

Purpose

Resumes error logging after an **errlast** command was issued.

Syntax

```
void errresume()
```

Description

When an error is logged with the **errlast** command, no more error logging will happen on the system until an **errresume** call is issued.

Execution Environment

This can be called from either the process or an interrupt level.

Related reference:

"errsave or errlast Kernel Service"

Related information:

Error-Logging Facility

errsave or errlast Kernel Service

Purpose

Allows the kernel and kernel extensions to write to the error log.

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/errids.h> void errsave (*buf*, *cnt***) char** **buf***;** **unsigned int** *cnt***; void errlast (***buf, cnt***) char** **buf* **unsigned int** *cnt***;**

Parameters

Item	Description
buf	Points to a buffer that contains an error record as described in the **/usr/include/sys/err_rec.h** file.
cnt	Specifies the number of bytes in the error record contained in the buffer pointed to by the *buf* parameter.

Description

The **errsave** kernel service allows the kernel and kernel extensions to write error log entries to the error device driver. The error record pointed to by the *buf* parameter includes the error ID resource name and detailed data.

In addition, the **errlast** kernel service disables any future error logging, thus any error logged with **errlast** will stay on NVRAM. This service is only for use prior to a pending system crash or stop. The **errlast** service should only be used in extreme circumstances where the system can not continue, such as the occurance of a machine check.

Execution Environment

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

Return Values

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

Related information:

errlog subroutine

Error Logging Special Files

RAS Kernel Services

f

The following kernel services begin with the with the letter f.

fetch_and_add Kernel Services

Purpose

Increments a variable atomically.

Syntax

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

int fetch_and_add (addr,  value)
atomic_p addr;
int value;

long fetch_and_addlp (addr,  value)
atomic_l addr;
long value;
```

Parameters

Item	Description
addr	Specifies the address of the variable to be incremented.
value	Specifies the value to be added to the variable.

Description

The **fetch_and_add** kernel services atomically increment a variable.

The **fetch_and_add** kernel service operates on a single word (32 bit) variable while the **fetch_and_addlp** kernel service operates on a double word (64 bit) variable.

These operations are useful when a counter variable is shared between several kernel threads, because it ensures that the fetch, update, and store operations used to increment the counter occur atomically (are not interruptible).

Note:

- The single word variable for the **fetch_and_add** kernel service must be aligned on a word (32 bit) boundary.
- The double word variable for the **fetch_and_addlp** kernel service must be aligned on a double word (64 bit) boundary.

Execution Environment

The **fetch_and_add** kernel services can be called from either the process or interrupt environment.

Return Values

The **fetch_and_add** kernel services return the original value of the variable.

Related reference:

"fetch_and_and or fetch_and_or Kernel Services"

"compare_and_swap Kernel Services" on page 45

Related information:

Locking Kernel Services

fetch_and_and or fetch_and_or Kernel Services

Purpose

Clears and sets bits in a variable atomically.

Syntax

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

uint fetch_and_and (addr,  mask)
atomic_p addr;uint mask;

ulong fetch_and_andlp (addr,  mask)
atomic_l addr;
ulong mask;

uint fetch_and_or (addr,  mask)
atomic_p addr;
uint mask;
```

```
ulong fetch_and_orlp (addr, mask)
atomic_l addr;
ulong mask;
```

Parameters

Item	Description
addr	Specifies the address of the variable whose bits are to be cleared or set.
mask	Specifies the bit mask which is to be applied to the variable.

Description

The **fetch_and_and** and **fetch_and_or** kernel services respectively clear and set bits in a variable, according to a bit mask, as a single atomic operation. The **fetch_and_and** service clears bits in the variable which correspond to clear bits in the bit mask, and the **fetch_and_or** service sets bits in the variable which correspond to set bits in the bit mask.

The **fetch_and_add** and **fetch_and_or** kernel services operate on a single word (32 bit) variable while the **fetch_and_addlp** and **fetch_and_orlp** kernel services operate on a double word (64 bit) variable.

These operations are useful when a variable containing bit flags is shared between several kernel threads, because they ensure that the fetch, update, and store operations used to clear or set a bit in the variable occur atomically (are not interruptible).

Note:

- For the **fetch_and_and** and **fetch_and_or** kernel services, the single word containing the bit flags must be aligned on a full word (32 bit) boundary.
- For the **fetch_and_addlp** and **fetch_and_orlp** kernel services, the double word containing the bit flags must be aligned on a double word (64 bit) boundary.

Execution Environment

The **fetch_and_and** and **fetch_and_or** kernel services can be called from either the process or interrupt environment.

Return Values

The **fetch_and_and** and **fetch_and_or** kernel services return the original value of the variable.

Related reference:

"fetch_and_add Kernel Services" on page 139

"compare_and_swap Kernel Services" on page 45

Related information:

Locking Kernel Services

fidtovp Kernel Service

Purpose

Maps a file system structure to a file ID.

Maps a file identifier to a mode.

Syntax

#include <sys/types.h> #include <sys/vnode.h> int fidtovp(*fsid*, *fid*, *vpp***) fsid_t ****fsid*; **struct fileid ****fid*; **struct vnode ****vpp*;

Parameters

Item	Description
fsid	Points to a file system ID structure. The system uses this structure to determine which virtual file system (VFS) contains the requested file.
fid	Points to a file ID structure. The system uses this pointer to locate the specific file within the VFS.
vpp	Points to a location to store the file's vnode pointer upon successful return of the **fidtovp** kernel service.

Description

The **fidtovp** kernel service returns a pointer to a vnode for the file identified by **fsid** and **fid**, and increments the count on the vnode so the file is not removed. Subroutines that call the **fidtovp** kernel service must call VNOP_RELE to release the vnode pointer.

This kernel service is designed for use by the server side of distributed file systems.

Execution Environment

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

Return Values

Item	Description
0	Indicates successful completion.
ESTALE	Indicates the requested file or file system was removed or recreated since last access with the given file system ID or file ID.

find_input_type Kernel Service

Purpose

Finds the given packet type in the Network Input Interface switch table and distributes the input packet according to the table entry for that type.

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <net/if.h> int find_input_type (*type*, *m*, *ac*, *header_pointer***) ushort** *type*; **struct mbuf *** *m*; **struct arpcom *** *ac*; **caddr_t** *header_pointer*;

Parameters

Item	Description
type	Specifies the protocol type.
m	Points to the **mbuf** buffer containing the packet to distribute.
ac	Points to the network common portion (**arpcom**) of the network interface on which the packet was received. This common portion is defined as follows: `in net/if_arp.h`
header_pointer	Points to the buffer containing the input packet header.

Description

The **find_input_type** kernel service finds the given packet type in the Network Input table and distributes the input packet contained in the **mbuf** buffer pointed to by the *m* value. The *ac* parameter is passed to services that do not have a queued interface.

Execution Environment

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

Return Values

Item	Description
0	Indicates that the protocol type was successfully found.
ENOENT	Indicates that the service could not find the type in the Network Input table.

Related reference:

"add_input_type Kernel Service" on page 8

"del_input_type Kernel Service" on page 64

Related information:

Network Kernel Services

fp_access Kernel Service

Purpose

Checks for access permission to an open file.

Syntax

#include <sys/types.h> #include <sys/errno.h> int fp_access (*fp*, *perm***) struct file ***fp***; int** *perm***;**

Parameters

Item	Description
fp	Points to a file structure returned by the **fp_open** or **fp_opendev** kernel service.
perm	Indicates which read, write, and execute permissions are to be checked. The **/usr/include/sys/mode.h** file contains pertinent values (IREAD, IWRITE, IEXEC).

Description

The **fp_access** kernel service is used to see if either the read, write, or exec bit is set anywhere in a file's permissions mode. Set *perm* to one of the following constants from **mode.h**:

IREAD IWRITE IEXEC

Execution Environment

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

Return Values

Item	Description
0	Indicates that the calling process has the requested permission.
EACCES	Indicates all other conditions.

Related information:

access subroutine

Logical File System Kernel Services

fp_close Kernel Service

Purpose

Closes a file.

Syntax

#include <sys/types.h> #include <sys/errno.h> int fp_close (*fp***) struct file** **fp***;**

Parameter

Item	Description
fp	Points to a file structure returned by the **fp_open**, **fp_getf**, or **fp_opendev** kernel service.

Description

The **fp_close** kernel service is a common service for closing files used by both the file system and routines outside the file system.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.
non-zero	The underlying file system implementation might report one of the values from the **/usr/include/errno.h** file, which is returned to the caller as a return value. However, the file is still closed.

Related information:

close subroutine

Logical File System Kernel Services

fp_close Kernel Service for Data Link Control (DLC) Devices

Purpose

Allows kernel to close the generic data link control (GDLC) device manager using a file pointer.

Syntax

int fp_close(*fp***)**

Parameters

Item	Description
fp	Specifies the file pointer of the GDLC being closed.

Description

The **fp_close** kernel service disables a GDLC channel. If this is the last channel to close on a port, the GDLC device manager resets to an idle state on that port and the communications device handler is closed. The **fp_close** kernel service may be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful completion.
ENXIO	Indicates an invalid file pointer. This value is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"fp_close Kernel Service" on page 144

"fp_open Kernel Service for Data Link Control (DLC) Devices" on page 156

Related information:

Generic Data Link Control (GDLC) Environment Overview

fp_fstat Kernel Service

Purpose

Gets the attributes of an open file.

Syntax

#include <sys/types.h> #include <sys/errno.h> int fp_fstat (*fp*, *statp*, *len*, *seg***) struct file*** *fp***; struct stat** **statp***; int** *len***; int** *seg***;**

Parameters

Item	Description
fp	Points to a file structure returned by the **fp_open** kernel service.
statp	Points to a buffer defined to be of **stat** or **fullstat** type structure. The *statsz* parameter indicates the buffer type.
len	Indicates the size of the **stat** or **fullstat** structure to be returned. The **/usr/include/sys/stat.h** file contains information about the **stat** structure.
seg	Specifies the flag indicating where the information represented by the *statbuf* parameter is located: **SYS_ADSPACE** Buffer is in kernel memory. **USER_ADSPACE** Buffer is in user memory.

Description

The **fp_fstat** kernel service is an internal interface to the function provided by the **fstatx** subroutine.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.

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

Related information:

fstatx subroutine

Logical File System Kernel Services

fp_fsync Kernel Service

Purpose

Writes changes for a specified range of a file to permanent storage.

Syntax

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

int fp_fsync (fp, how, off, len)
struct file *fp;
int how;
offset_t off;
offset_t len;
```

Description

The **fp_fsync** kernel service is an internal interface to the function provided by the **fsync_range** subroutine.

Parameters

Item — **Description**

fp
Points to a file structure returned by the **fp_open** kernel service.

how
Specifies the following handling characteristics of the operation:

FDATASYNC
The changed data in the range specified by the `off` and `len` parameters is written to the storage. If the metadata for the file is changed and this changed metadata must read the data, the metadata is also written to the storage. Otherwise, the metadata is not updated.

FFILESYNC
The changed data in the range specified by the `off` and `len` parameters is written to the storage. If any metadata is changed, all of the changed user data is written to the storage. Metadata changes and file attributes including time stamps are also written to the storage.

off
Specifies the starting offset value of the data in the file to be written to the storage.

len
Specifies the length of the file range to be written to the storage. If you specify the value as zero, all cached data is written to the storage.

Execution Environment

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

Return Values

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

Related information:

fsync or fsync_range Subroutine

Logical File System Kernel Services

fp_getdevno Kernel Service

Purpose

Gets the device number or channel number for a device.

Syntax

#include <sys/types.h> #include <sys/errno.h> #include <sys/file.h> int fp_getdevno (*fp*, *devp*, *chanp***)** **struct file** **fp***;** **dev_t** **devp***;** **chan_t** **chanp***;**

Parameters

Item	Description
fp	Points to a file structure returned by the **fp_open** or **fp_opendev** service.
devp	Points to a location where the device number is to be returned.
chanp	Points to a location where the channel number is to be returned.

Description

The **fp_getdevno** service finds the device number and channel number for an open device that is associated with the file pointer specified by the *fp* parameter. If the value of either *devp* or *chanp* parameter is null, this service does not attempt to return any value for the argument.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates that the pointer specified by the *fp* parameter does not point to a file structure for an open device.

Related information:

Logical File System Kernel Services

fp_getea Kernel Service

Purpose

Reads the value of an extended attribute value.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
int fp_getea (fp,name, value, size, countp, segflag)
struct file * fp;
const char * name;
```

```
void * value;
size_t size;
ssize_t * countp;
int segflag;
```

Parameters

Item	Description
fp	Specifies the file structure returned by the **fp_open** kernel service.
name	Specifies the name of the extended attribute. An extended attribute name is a NULL-terminated string.
value	Specifies the pointer to a buffer in which the attribute is stored. The value of an extended attribute is an opaque byte stream of specified length.
size	Specifies the size of the value buffer. If size is 0, **fp_getea** returns the current size of the named extended attribute, which can be used to estimate whether the size of a buffer is sufficiently large to hold the value associated with the extended attribute.
countp	Specifies the actual size of the content in the value buffer.
segflag	Specifies the flag indicating where the pointer specified by the path parameter is located: **SYS_ADSPACE** The pointers specified by the *name* and *value* parameters are stored in kernel memory. **USER_ADSPACE** The pointers specified by the *name* and *value* parameters are stored in application memory.

Description

The **fp_getea** kernel service provides a common service used by:

- The file system for the implementation of the **fgetea** subroutine.
- Kernel routines outside the file system that set extended attribute values.

Execution Environment

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

Return Values

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

Related information:

fgetea subroutine

fp_open subroutine

Logical File System Kernel Services

fp_getf Kernel Service

Purpose

Retrieves a pointer to a file structure.

Syntax

#include <sys/types.h> #include <sys/errno.h> int fp_getf (*fd***,** *fpp***) int** *fd***; struct file** ***fpp***;**

Parameters

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

Description

A process calls the **fp_getf** kernel service when it has a file descriptor for an open file, but needs a file pointer to use other Logical File System services.

The **fp_getf** kernel service uses the file descriptor as an index into the process's open file table. From this table it extracts a pointer to the associated file structure.

As a side effect of the call to the **fp_getf** kernel service, the reference count on the file descriptor is incremented. This count must be decremented when the caller has completed its use of the returned file pointer. The file descriptor reference count is decremented by a call to the **ufdrele** kernel service.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.
EBADF	Indicates that either the file descriptor is invalid or not currently used in the process.

Related reference:

"ufdhold and ufdrele Kernel Service" on page 515

Related information:

Logical File System Kernel Services

fp_get_path Kernel Service

Purpose

Returns the full path name of the file referenced by the **fp** parameter.

Syntax

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

int
fp_get_path(struct file     *fp,
            int             flags,
            char            *path,
            size_t          size)
```

Parameters

fp Points to a file structure that is returned by the **fp_open** or **fp_opendev** kernel service.

flags
No flags are defined; this parameter must be 0.

path
Points to a buffer where the file name is returned.

size
Specifies the size of the path buffer.

Description

The **fp_get_path** kernel service provides a method to find a path name from a file structure pointer.

Execution environment

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

Return values

0 Indicates a successful operation.

EINVAL
Invalid **fp** or **path** argument, or the **fp** parameter does not refer to a DTYPE_VNODE file structure.

fp_hold Kernel Service

Purpose

Increments the open count for a specified file pointer.

Syntax

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

void fp_hold ( fp)
struct file *fp;
```

Parameter

Item	Description
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.

Related information:

Logical File System Kernel Services

fp_ioctl Kernel Service

Purpose

Issues a control command to an open device or file.

Syntax

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

int fp_ioctl (fp, cmd, arg, ext)
struct file * fp;
unsigned longcmd;
caddr_targ;
intext;
```

Parameters

Item	Description
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

Item	Description
0	Indicates a successful operation.

If an error occurs, one of the values from the **/usr/include/sys/errno.h** file is returned. The **ioctl** subroutine contains valid **errno** values.

Related information:

ioctl subroutine

Logical File System Kernel Services

fp_ioctl Kernel Service for Data Link Control (DLC) Devices

Purpose

Transfers special commands from the kernel to generic data link control (GDLC) using a file pointer.

Syntax

```
#include <sys/gdlextcb.h>
#include <fcntl.h>
int fp_ioctl (fp, cmd, arg, ext)
```

Parameters

Item	Description
fp	Specifies the file pointer of the target GDLC.
cmd	Specifies the operation to be performed by GDLC. For a listing of all possible operators, see "ioctl Operations (op) for DLC"*Technical Reference: Communications, Volume 1*.
arg	Specifies the address of the parameter block. The argument for this parameter must be in the kernel space. For a listing of possible values, see "Parameter Blocks by ioctl Operation for DLC"*Technical Reference: Communications, Volume 1*.
ext	Specifies the extension parameter. This parameter is ignored by GDLC.

Description

Various GDLC functions can be initiated using the **fp_ioctl** kernel service, such as changing configuration parameters, contacting the remote, and testing a link. Most of these operations can be completed before returning to the user synchronously. Some operations take longer, so asynchronous results are returned much later using the **exception** function handler. GDLC calls the kernel user's exception handler to complete these results. Each GDLC supports the **fp_ioctl** kernel service by way of its **dlcioctl** entry point. The **fp_ioctl** kernel service may be called from the process environment only.

Note: The **DLC_GET_EXCEP** ioctl operation is not used since all exception conditions are passed to the kernel user through the exception handler.

Return Values

Item	Description
0	Indicates a successful completion.
ENXIO	Indicates an invalid file pointer.
EINVAL	Indicates an invalid value.
ENOMEM	Indicates insufficient resources to satisfy the **ioctl** subroutine.

These return values are defined in the **/usr/include/sys/errno.h** file.

Related reference:

“fp_ioctl Kernel Service” on page 150

Related information:

ioctl subroutine

Generic Data Link Control (GDLC) Environment Overview

fp_ioctlx Kernel Service

Purpose

Issues a control command to an open device.

Syntax

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

int fp_ioctlx (fp, cmd, arg, ext, flags, retval)
struct file *fp;
unsigned long cmd;
caddr_t arg;
ext_t ext;
unsigned long flags;
long *retval;
```

Description

The **fp_ioctlx** kernel service is an internal interface to the function provided by the **ioctl** subroutine.

The **fp_ioctlx** kernel service issues a control command to an open device. Some drivers need the return value that is returned by the kernel service if there is no error. This value is not available through the **fp_ioctl** kernel service. The **fp_ioctlx** kernel service allows this data to be passed.

Parameters

Item	Description
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.
flags	Indicates the address space of *arg* parameter. If the *arg* value is in kernel address space, *flags* should be specified as **FKERNEL**. Otherwise, it should be zero (drivers pass data that is in user space).
retval	Points to the location where the return value will be stored on successful return from the call.

Execution Environment

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

Return Values

Upon successful completion, the **fp_ioctlx** kernel service returns 0. If unsuccessful, one of the values from the **/usr/include/sys/errno.h** file is returned. The **ioctl** subroutine contains valid **errno** values. This value will be stored in the *retval* parameter.

Related reference:

"fp_ioctl Kernel Service" on page 150

Related information:

ioctl, ioctlx, ioctl32, or ioctl32x Subroutine

fp_listea Kernel Service

Purpose

Lists the extended attributes associated with a file.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
int fp_listea (fp, list, size, countp, segflag)
struct file * fp;
const char * list;
size_t size;
ssize_t * countp;
int segflag;
```

Parameters

Item	Description
fp	Specifies the file structure returned by the **fp_open** kernel service.
list	Specifies a pointer to a buffer in which the list of attributes will be stored.
size	Specifies the size of the list buffer.
countp	Specifies the actual size of the content in the list buffer.
segflag	Specifies the flags indicating where the pointer specified by the path parameter is located: **SYS_ADSPACE** The pointer specified by the list parameter is stored in kernel memory. **USER_ADSPACE** The pointer specified by the list parameter is stored in application memory.

Description

The **fp_listea** kernel service provides a common service used by:

- File system for the implementation of the **flistea** subroutine.
- Kernel routines outside the file system that set extend attribute values.

Execution Environment

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

Return Values

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

Related information:

flistea subroutine

fp_open subroutine

Logical File System Kernel Services

fp_lseek, fp_llseek Kernel Service

Purpose

Changes the current offset in an open file.

Syntax

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

int fp_lseek  ( fp,  offset,  whence)
struct file *fp;
off_t offset;
int whence;

int fp_llseek
( fp,  offset,  whence)
struct file *fp
offset_t offset;
int whence;
```

Parameters

Item	Description
fp	Points to a file structure returned by the **fp_open** kernel service.
offset	Specifies the number of bytes (positive or negative) to move the file pointer.
whence	Indicates how to use the offset value: **SEEK_SET** Sets file pointer equal to the number of bytes specified by the *offset* parameter. **SEEK_CUR** Adds the number of bytes specified by the *offset* parameter to current file pointer. **SEEK_END** Adds the number of bytes specified by the *offset* parameter to current end of file.

Description

The **fp_lseek** and **fp_llseek** kernel services are internal interfaces to the function provided by the **lseek** and **llseek** subroutines.

Execution Environment

The **fp_lseek** and **fp_llseek** kernel services can be called from the process environment only.

Return Values

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

Related information:

lseek subroutine

Logical File System Kernel Services

fp_open Kernel Service

Purpose

Opens special and regular files or directories.

Syntax

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

int fp_open (path,oflags,mode,ext,segflag,fpp)
char * path;
long  oflags;
int  mode;
ext_t  ext;
int  segflag;
struct file ** fpp;
```

Parameters

Item	Description
path	Points to the file name of the file to be opened.
oflags	Specifies open mode flags as described in the **open** subroutine.
mode	Specifies the mode (permissions) value to be given to the file if the file is to be created.
ext	Specifies an extension argument required by some device drivers. Individual drivers determine its content, form, and use.
segflag	Specifies the flag indicating where the pointer specified by the *path* parameter is located: **SYS_ADSPACE** The pointer specified by the *path* parameter is stored in kernel memory. **USER_ADSPACE** The pointer specified by the *path* parameter is stored in application memory.
fpp	Points to the location where the file structure pointer is to be returned by the **fp_open** service.

Description

The **fp_open** kernel service provides a common service used by:

- The file system for the implementation of the **open** subroutine
- Kernel routines outside the file system that must open files

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.

Also, the *fpp* parameter points to an open file structure that is valid for use with the other Logical File System services. If an error occurs, one of the values from the **/usr/include/sys/errno.h** file is returned. The discussion of the **open** subroutine contains possible **errno** values.

Related information:

open subroutine

Logical File System Kernel Services

fp_open Kernel Service for Data Link Control (DLC) Devices

Purpose

Allows kernel to open the generic data link control (GDLC) device manager by its device name.

Syntax

```
#include <sys/gdlextcb.h>
#include <fcntl.h>
```

```
fp_open (path, oflags, cmode, ext, segflag, fpp)
```

Parameters

Item	Description
path	Consists of a character string containing the **/dev** special file name of the GDLC device manager, with the name of the communications device handler appended. The format is shown in the following example: `/dev/dlcether/ent0`
oflags	Specifies a value to set the file status flag. The GDLC device manager ignores all but the following values: **O_RDWR** Open for reading and writing. This must be set for GDLC or the open will not be successful. **O_NDELAY, O_NONBLOCK** Subsequent writes return immediately if no resources are available. The calling process is not put to sleep.
cmode	Specifies the **O_CREAT** mode parameter. This is ignored by GDLC.
ext	Specifies the extended kernel service parameter. This is a pointer to the **dlc_open_ext** extended I/O structure for open subroutines. The argument for this parameter must be in the kernel space. "open Subroutine Extended Parameters for DLC"*Technical Reference: Communications, Volume 1*provides more information on the extension parameter.
segflag	Specifies the segment flag indicating where the *path* parameter is located: **FP_SYS** The *path* parameter is stored in kernel memory. **FP_USR** The *path* parameter is stored in application memory.
fpp	Specifies the returned file pointer. This parameter is passed by reference and updated by the file I/O subsystem to be the file pointer for this **open** subroutine.

Description

The **fp_open** kernel service allows the kernel user to open a GDLC device manager by specifying the special file names of both the DLC and the communications device handler. Since the GDLC device manager is multiplexed, more than one process can open it (or the same process multiple times) and still have unique channel identifications.

Each open carries the communications device handler's special file name so that the DLC knows which port to transfer data on.

The kernel user must also provide functional entry addresses in order to obtain receive data and exception conditions. Each GDLC supports the **fp_open** kernel service via its **dlcopen** entry point. The **fp_open** kernel service may be called from the process environment only. "Using GDLC Special Kernel Services" in *AIX Version 6.1 Communications Programming Concepts* provides additional information.

Return Values

Upon successful completion, this service returns a value of 0 and a valid file pointer in the *fpp* parameter.

Item	Description
ECHILD	Indicates that the service cannot create a kernel process.
EINVAL	Indicates an invalid value.
ENODEV	Indicates that no such device handler is present.
ENOMEM	Indicates insufficient resources to satisfy the open.
EFAULT	Indicates that the kernel service, such as the **copyin** or **initp** service, has failed.

These return values are defined in the **/usr/include/sys/errno.h** file.

Related reference:

“fp_open Kernel Service” on page 155

“fp_close Kernel Service for Data Link Control (DLC) Devices” on page 144

Related information:

open Subroutine Extended Parameters for DLC

Generic Data Link Control (GDLC) Environment Overview

fp_opendev Kernel Service

Purpose

Opens a device special file.

Syntax

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

int fp_opendev (devno,flags,cname, ext, fpp)
dev_tdevno;
int  flags;
caddr_t cname;
ext_t  ext;
struct file** fpp;
```

Parameters

Item	Description
devno	Specifies the major and minor device number of device driver to open.
flag	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.
cname	Points to a channel specifying a character string or a null value.
ext	Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.
fpp	Specifies the returned file pointer. This parameter is passed by reference and is updated by the **fp_opendev** service to be the file pointer for this open instance. This file pointer is used as input to other Logical File System services to specify the open instance.

Description

The kernel or kernel extension calls the **fp_opendev** kernel service to open a device by specifying its device major and minor number. The **fp_opendev** kernel service provides the correct semantics for opening the character or multiplexed class of device drivers.

If the specified device driver is non-multiplexed:

- An in-core i-node is found or created for this device.
- The i-node reference count is increment by 1.
- The device driver's **ddopen** entry point is called with the *devno*, *devflag*, and *ext* parameters. The unused *chan* parameter on the call to the **ddopen** routine is set to 0.

If the device driver is a multiplexed character device driver (that is, its **ddmpx** entry point is defined), an in-core i-node is created for this channel. The device driver's **ddmpx** routine is also called with the *cname* pointer to the channel identification string if non-null. If the *cname* 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 **fpopendev** kernel service returns an **ENXIO** error code. If successful, the i-node reference count is increment by 1. The device driver **ddopen** routine is also called with the *devno*, *flag*, *chan* (provided by **ddmpx** routine), and *ext* parameters.

If the return value from the specified device driver **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 **ddopen** routine is 0, the **fp_opendev** service returns the file pointer corresponding to this open of the device.

The **fp_opendev** kernel service can only be called in the process environment or device driver top half. Interrupt handlers cannot call it. It is assumed that all arguments to the **fp_opendev** kernel service are in kernel space.

The file pointer (*fpp*) returned by the **fp_opendev** kernel service is only valid for use with a subset of the Logical File System services. These nine services can be called:

- **fp_close**
- **fp_ioctl**
- **fp_poll**
- **fp_select**
- **fp_read**
- **fp_readv**
- **fp_rwuio**
- **fp_write**
- **fp_writev**

Other services return an **EINVAL** return value if called.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.

The **fpp* field also points to an open file structure that is valid for use with the other Logical File System services. If an error occurs, one of the following values from the **/usr/include/sys/errno.h** file is returned:

Item	Description
EINVAL	Indicates that the major portion of the *devno* parameter exceeds the maximum number allowed, or the *flags* parameter is not valid.
ENODEV	Indicates that the device does not exist.
EINTR	Indicates that the signal was caught while processing the **fp_opendev** request.
ENFILE	Indicates that the system file table is full.
ENXIO	Indicates that the device is multiplexed and unable to allocate the channel.

The **fp_opendev** service also returns any nonzero return code returned from a device driver **ddopen** routine.

Related reference:

"ddopen Device Driver Entry Point" on page 628

"fp_close Kernel Service" on page 144

Related information:

Logical File System Kernel Services

fp_poll Kernel Service

Purpose

Checks the I/O status of multiple file pointers, file descriptors, and message queues.

Syntax

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

int fp_poll (listptr, nfdsmsgs, timeout, flags)
void * listptr;
unsigned long  nfdsmsgs;
long  timeout;
uint  flags;
```

Parameters

Item	Description
listptr	Points to an array of **pollfd** or **pollmsg** structures, or to a single **pollist** structure. Each structure specifies a file pointer, file descriptor, or message queue ID. The events of interest for this file or message queue are also specified.
nfdsmsgs	Specifies the number of files and message queues to check. The low-order 16 bits give the number of elements present in the array of **pollfd** structures. The high-order 16 bits give the number of elements present in the array of **pollmsg** structures. If either half of the *nfdsmsgs* parameter is equal to 0, then the corresponding array is presumed abse1e.
timeout	Specifies how long the service waits for a specified event to occur. If the value of this parameter is -1, the **fp_poll** kernel service does not return until at least one of the specified events has occurred. If the time-out value is 0, the **fp_poll** kernel service does not wait for an event to occur. Instead, the service returns immediately even if none of the specified events have occurred. For any other value of the *timeout* parameter, the **fp_poll** kernel service specifies the maximum length of time (in milliseconds) to wait for at least one of the specified events to occur.

Item	Description
flags	Specifies the type of data in the *listptr* parameter: **POLL_FDMSG** Input is a file descriptor and/or message queue. **0** Input is a file pointer.

Description

Note: The **fp_poll** service applies only to character devices, pipes, message queues, and sockets. Not all character device drivers support the **fp_poll** service.

The **fp_poll** kernel service checks the specified file pointers/descriptors and message queues to see if they are ready for reading or writing, or if they have an exceptional condition pending.

The **pollfd**, **pollmsg**, and **pollist** structures are defined in the **/usr/include/sys/poll.h** file. These are the same structures described for the **poll** subroutine. One difference is that the **fd** field in the **pollfd** structure contains a file pointer when the *flags* parameter on the **fp_poll** kernel service equals 0 (zero). If the *flags* parameter is set to a **POLL_FDMSG** value, the field is taken as a file descriptor in all processed **pollfd** structures. If either the **fd** or **msgid** fields in their respective structures has a negative value, the processing for that structure is skipped.

When performing a poll operation on both files and message queues, the *listptr* parameter points to a **pollist** structure, which can specify both files and message queues. To construct a **pollist** structure, use the **POLLIST** macro as described in the **poll** subroutine.

If the number of **pollfd** elements in the *nfdsmsgs* parameter is 0, then the *listptr* parameter must point to an array of **pollmsg** structures.

If the number of **pollmsg** elements in the *nfdsmsgs* parameter is 0, then the *listptr* parameter must point to an array of **pollfd** structures.

If the number of **pollmsg** and **pollfd** elements are both nonzero in the *nfdsmsgs* parameter, the *listptr* parameter must point to a **pollist** structure as previously defined.

Execution Environment

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

Return Values

Upon successful completion, the **fp_poll** kernel service returns a value that indicates the total number of files and message queues that satisfy the selection criteria. The return value is similar to the *nfdsmsgs* parameter in the following ways:

- The low-order 16 bits give the number of files.
- The high-order 16 bits give the number of message queue identifiers that have nonzero *revents* values.

Use the **NFDS** and **NMSGS** macros to separate these two values from the return value. A return code of 0 (zero) indicates that:

- The call has timed out.
- None of the specified files or message queues indicates the presence of an event.

In other words, all *revents* fields are 0 (zero).

When the return code from the **fp_poll** kernel service is negative, it is set to the following value:

Item	Description
EINTR	Indicates that a signal was caught during the **fp_poll** kernel service.

Related reference:

"selreg Kernel Service" on page 464

Related information:

poll subroutine

Logical File System Kernel Services

fp_read Kernel Service

Purpose

Performs a read on an open file with arguments passed.

Syntax

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

int fp_read (fp, buf, nbytes, ext, segflag, countp)
struct file * fp;
char * buf;
ssize_t  nbytes;
ext_t   ext;
int  segflag;
ssize_t * countp;
```

Parameters

Item	Description
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

Item	Description
0	Indicates successful completion.

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

Related information:

read subroutine

Logical File System Kernel Services

fp_readv Kernel Service

Purpose

Performs a read operation on an open file with arguments passed in **iovec** elements.

Syntax

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

int fp_readv
(fp, iov, iovcnt, ext,
seg, countp)
struct file * fp;
struct iovec * iov;
ssize_t  iovcnt;
ext_t   ext;
int  seg;
ssize_t countp;
```

Parameters

Item	Description
fp	Points to a file structure returned by the **fp_open** kernel service.
iov	Points to an array of **iovec** elements. Each **iovec** element describes a buffer where data to be read from the file is to be stored.
iovcnt	Specifies the number of **iovec** elements in the array pointed to by the *iov* parameter.
ext	Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.
seg	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

Item	**Description**
0	Indicates a successful operation.

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

Related information:

readv subroutine

Logical File System Kernel Services

fp_removeea Kernel Service

Purpose

Removes an extended attribute.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
int fp_removeea (fp, name, segflag)
struct file * fp;
const char * name;
int segflag;
```

Parameters

Item	**Description**
fp	Specifies a file structure returned by the **fp_open** kernel service.
name	Specifies the name of the extended attribute. An extended attribute name is a NULL-terminated string.
segflag	Specifies the flag indicating where the pointer specified by the path parameter is located: **SYS_ADSPACE** The pointer specified by the *name* parameter is stored in kernel memory. **USER_ADSPACE** The pointer specified by the *name* parameter is stored in application memory.

Description

The **fp_removeea** kernel service provides a common service used by:

- The file system for the implementation of the **fremoveea** subroutine
- Kernel routines outside the file system that set extended attribute values

Execution Environment

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

Return Values

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

Related information:

fremoveea subroutine

fp_open subroutine

Logical File System Kernel Services

fp_rwuio Kernel Service

Purpose

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

Syntax

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

int fp_rwuio
( fp,  rw,  uiop,  ext)
struct file *fp;
enum uio_rw rw;
struct uio *uiop;
ext_t ext;
```

Parameters

Item	Description
fp	Points to a file structure returned by the **fp_open** or **fp_opendev** kernel service.
rw	Indicates whether this is a read operation or a write operation. It has a value of **UIO_READ** or **UIO_WRITE**.
uiop	Points to a **uio** structure, which contains information such as where to move data and how much to move.
ext	Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.

Description

The **fp_rwuio** kernel service is not the preferred interface for read and write operations. The **fp_rwuio** kernel service should only be used if the calling routine has been passed a **uio** structure. If the calling routine has not been passed a **uio** structure, it should not attempt to construct one and call the **fp_rwuio** kernel service with it. Rather, it should pass the requisite **uio** components to the **fp_read** or **fp_write** kernel services.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.

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

Related reference:

"uio Structure" on page 638

Related information:

Logical File System Kernel Services

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

Item **Description**

fp Points to the open instance of the device driver, socket, or pipe for which the low-level select operation is intended.

events Identifies the events that are to be checked. There are three standard event flags defined for the **poll** and **select** functions and one informational flag. The **/usr/include/sys/poll.h** file details the event bit definition. The four basic indicators are:

POLLIN
Input is present for the specified object.

POLLOUT
The specified file object is capable of accepting output.

POLLPRI
An exception condition has occurred on the specified object.

POLLSYNC
This is a synchronous request only. If none of the requested events are true, the selected routine should not remember this request as pending. That is, the routine does not need to call the **selnotify** service because of this request.

rtneventp Indicates the returned events pointer. This parameter, passed by reference, is used to indicate which selected events are true at the current time. The returned event bits include the requested events plus an additional error event indicator:

POLLERR
An error condition was indicated by the object's select routine. If this flag is set, the nonzero return code from the specified object's select routine is returned as the return code from the **fp_select** kernel service.

notify Points to a routine to be called when the specified object invokes the **selnotify** kernel service for an outstanding asynchronous select or poll event request. If no routine is to be called, this parameter must be NULL.

Description

The **fp_select** kernel service is a low-level service used by kernel extensions to perform a select operation for an open device, socket, or named pipe. The **fp_select** kernel service can be used for both synchronous and asynchronous select requests. Synchronous requests report on the current state of a device, and asynchronous requests allow the caller to be notified of future events on a device.

Invocation from a Device Driver's ddselect Routine

A device driver's **ddselect** routine can call the **fp_select** kernel service to pass select/poll requests to other device drivers. The **ddselect** routine for one device invokes the **fp_select** kernel service, which calls the **ddselect** routine for a second device, and so on. This is required when event information for the original device depends upon events occurring on other devices. A cascaded chain of select requests can be initiated that involves more than two devices, or a single device can issue **fp_select** calls to several other devices.

Each **ddselect** routine should preserve, in its call to the **fp_select** kernel service, the same **POLLSYNC** indicator that it received when previously called by the **fp_select** kernel service.

Invocation from Outside a Device Driver's ddselect Routine

If the **fp_select** kernel service is invoked outside of the device driver's **ddselect** routine, the **fp_select** kernel service sets the **POLLSYNC** flag, always making the request synchronous. In this case, no notification of future events for the specified device occurs, nor is a **notify** routine called, if specified. The **fp_select** kernel service can be used in this manner (unrelated to a poll or select request in progress) to check an object's current status.

Asynchronous Processing and the Use of the notify Routine

For asynchronous requests, the **fp_select** kernel service allows its callers to register a **notify** routine to be called by the kernel when specified events become true. When the relevant device driver detects that one or more pending events have become true, it invokes the **selnotify** kernel service. The **selnotify** kernel service then calls the **notify** routine, if one has been registered. Thus, the **notify** routine is called at interrupt time and must be programmed to run in an interrupt environment.

Use of a **notify** routine affects both the calling sequence at interrupt time and how the requested information is actually reported. Generalized asynchronous processing entails the following sequence of events:

1. A select request is initiated on a device and passed on (by multiple **fp_select** kernel service invocations) to further devices. Eventually, a device driver's **ddselect** routine that is not dependent on other devices for information is reached. This **ddselect** routine finds that none of the requested events are true, but remembers the asynchronous request, and returns to the caller. In this way, the entire chain of calls is backed out, until the origin of the select request is reached. The kernel then puts the originating process to sleep.
2. Later, one or more events become true for the device remembering the asynchronous request. The device driver routine (possibly an interrupt handler) calls the **selnotify** kernel service.
3. If the events are still being waited on, the **selnotify** kernel service responds in one of two ways. If no **notify** routine was registered when the select request was made for the device, then all processes waiting for events on this device are awakened. If a **notify** routine exists for the device, then this routine is called. The **notify** routine determines whether the original requested event should be reported as true, and if so, calls the **selnotify** kernel service on its own.

The following example details a cascaded scenario involving several devices. Suppose that a request has been made for Device A, and Device A depends on Device B, which depends on Device C. When specified events become true at Device C, the **selnotify** kernel service called from Device C's device driver performs differently depending on whether a **notify** routine was registered at the time of the request.

Cascaded Processing without the Use of notify Routines

If no **notify** routine was registered from Device B, then the **selnotify** kernel service determines that the specified events are to be considered true for the device driver at the head of the cascading chain. (The head of the chain, in this case Device A, is the first device driver to issue the **fp_select** kernel service from its select routine.) The **selnotify** kernel service awakens all processes waiting for events that have occurred on Device A.

It is important to note that when no **notify** routine is used, any device driver in the calling chain that reports an event with the **selnotify** kernel service causes that event to appear true for the first device in the chain. As a result, any processes waiting for events that have occurred on that first device are awakened.

Cascaded Processing with notify Routines

If, on the other hand, **notify** routines have been registered throughout the chain, then each interrupting device (by calling the **selnotify** kernel service) invokes the **notify** routine for the device above it in the calling chain. Thus in the preceding example, the **selnotify** kernel service for Device C calls the **notify** routine registered when Device B's **ddselect** routine invoked the **fp_select** kernel service. Device B's **notify** routine must then decide whether to again call the **selnotify** kernel service to alert Device A's **notify** routine. If so, then Device A's **notify** routine is called, and makes its own determination whether to call another **selnotify** routine. If it does, the **selnotify** kernel service wakes up all the processes waiting on occurred events for Device A.

A variation on this scenario involves a cascaded chain in which only some device drivers have registered **notify** routines. In this case, the **selnotify** kernel service at each level calls the **notify** routine for the level above, until a level is encountered for which no **notify** routine was registered. At this point, all events of interest are determined to be true for the device driver at the head of the cascading chain. If any **notify** routines were registered in levels above the current level, they are never called.

Returning from the fp_select Kernel Service

The **fp_select** kernel service does not wait for any selected events to become true, but returns immediately after the call to the object's **ddselect** routine has completed.

If the object's select routine is successfully called, the return code for the **fp_select** kernel service is set to the return code provided by the object's **ddselect** routine.

Execution Environment

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

Return Values

Item	Description
0	Indicates successful completion.
EAGAIN	Indicates that the allocation of internal data structures failed. The *rtneventp* parameter is not updated.
EINVAL	Indicates that the *fp* parameter is not a valid file pointer. The *rtneventp* parameter has the **POLLNVAL** flag set.

The **fp_select** kernel service can also be set to the nonzero return code from the specified object's **ddselect** routine. The *rtneventp* parameter has the **POLLERR** flag set.

Related reference:

"fp_poll Kernel Service" on page 160

"fp_select Kernel Service notify Routine" on page 169

Related information:

select subroutine

fp_select Kernel Service notify Routine

Purpose

Registers the **notify** routine.

Syntax

#include <sys/types.h> #include <sys/errno.h> void notify (*id*, *sub_id*, *rtnevents*, *pid***) int** *id***; int** *sub_id* **; ushort** *rtnevents* **; pid_t** *pid***;**

Parameters

Item	Description
id	Indicates the selected function ID specified by the routine that made the call to the **selnotify** kernel service to indicate the occurrence of an outstanding event. For device drivers, this parameter is equivalent to the *devno* (device major and minor number) parameter.
sub_id	Indicates the unique ID specified by the routine that made the call to the **selnotify** kernel service to indicate the occurrence of an outstanding event. For device drivers, this parameter is equivalent to the *chan* parameter: channel for multiplexed drivers; 0 for nonmultiplexed drivers.
rtnevents	Specifies the *rtnevents* parameter supplied by the routine that made the call to the **selnotify** service indicating which events are designated as true.
pid	Specifies the process ID of a process waiting for the event corresponding to this call of the **notify** routine.

When a **notify** routine is provided for a cascaded function, the **selnotify** kernel service calls the specified **notify** routine instead of posting the process that was waiting on the event. It is up to this **notify** routine to determine if another **selnotify** call should be made to notify the waiting process of an event.

The **notify** routine is not called if the request is synchronous (that is, if the **POLLSYNC** flag is set in the *events* parameter) or if the original poll or select request is no longer outstanding.

Note: When more than one process has requested notification of an event and the **fp_select** kernel service is used with a **notify** routine specified, the notification of the event causes the **notify** routine to be called once for each process that is currently waiting on one or more of the occurring events.

Description

The **fp_select** kernel service **notify** routine is registered by the caller of the **fp_select** kernel service to be called by the kernel when specified events become true. The option to register this **notify** routine is available in a cascaded environment. The **notify** routine can be called at interrupt time.

Execution Environment

The **fp_select** kernel service **notify** routine can be called from either the process or interrupt environment.

Related reference:

"fp_select Kernel Service" on page 166

"selnotify Kernel Service" on page 462

Related information:

Logical File System Kernel Services

fp_setea Kernel Service

Purpose

Sets an extended attribute value.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
int fp_setea (fp, name, value, size, flags, segflag)
struct file *fp;
const char * name;
void * value;
size_t size;
intflags
int segflag;
```

Parameters

Item	Description
fp	Specifies a file structure returned by the **fp_open** kernel service.
name	Specifies the name of the extended attribute. An extended attribute name is a NULL terminated string.
value	Specifies a pointer to the value of an attribute. The value of an extended attribute is an opaque byte stream of specified length.
size	Specifies the length of the value.
flags	None of the flags are defined at this time.
segflag	Specifies the flag indicating where the pointer specified by the *path* parameter is located: **SYS_ADSPACE** The pointers specified by the *name* and *value* parameters are stored in kernel memory. **USER_ADSPACE** The pointers specified by the *name* and *value* parameters are stored in application memory.

Description

The **fp_setea** kernel service provides a common service used by the following routines:

- The file system for the implementation of the **fsetea** subroutine
- Kernel routines outside the file system that set extended attribute values

Execution Environment

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

Return Values

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

Related information:

fsetea subroutine

fp_open subroutine

Logical File System Kernel Services

fp_statea Kernel Service

Purpose

Provides information on an extended attribute.

Syntax

```
#include  <sys/types.h>
#include <sys/errno.h>
int fp_statea ( fp, name, buffer, segflag)
struct file * fp;
const char * name;
struct stat64x * buffer;
int segflag;
```

Parameters

Item	Description
fp	Specifies a file structure returned by the **fp_open** kernel service.
name	Specifies the name of the extended attribute. An extended attribute name is a NULL terminated string.
buffer	Specifies a pointer to the **stat** structure in which information is returned.
segflag	Specifies the flag indicating the location of the pointer stored by the *path* parameter is located: **SYS_ADSPACE** The pointers specified by the *name* and *value* parameters are stored in kernel memory. **USER_ADSPACE** The pointers specified by the *name* and *value* parameters are stored in application memory.

Description

The **fp_statea** kernel service provides a common service used by the following routines:

- The file system for the implementation of the **fstatea** subroutine
- Kernel routines outside the file system that set extended attribute values.

Execution Environment

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

Return Values

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

Related information:

fstatea subroutine

Logical File System Kernel Services

fp_write Kernel Service

Purpose

Performs a write operation on an open file with arguments passed.

Syntax

#include <sys/types.h> #include <sys/errno.h> int fp_write (*fp*, *buf*, *nbytes*, *ext*, *seg*, *countp***) struct file *** *fp*; **char *** *buf*; **ssize_t** *nbytes*, **ext_t** *ext*; **int** *seg*; **ssize_t *** *countp*;

Parameters

Item	Description
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.
seg	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

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

Related information:

write subroutine

Logical File System Kernel Services

fp_write Kernel Service for Data Link Control (DLC) Devices

Purpose

Allows kernel data to be sent using a file pointer.

Syntax

#include <sys/gdlextcb.h> #include <sys/fp_io.h> int fp_write (*fp***,** *buf***,** *nbytes***,** *ext***,** *segflag***,** *countp***)**

Parameters

Item	Description
fp	Specifies file pointer returned from the **fp_open** kernel service.
buf	Points to a kernel **mbuf** structure.
nbytes	Contains the byte length of the write data. It is not necessary to set this field to the actual length of write data, however, since the **mbuf** contains a length field. Instead, this field can be set to any non-negative value (generally set to 0).
ext	Specifies the extended kernel service parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for writes. The argument for this parameter must be in the kernel space. For more information on this parameter, see "write Subroutine Extended Parameters for DLC"*Technical Reference: Communications, Volume 1*.

Item	Description
segflag	Specifies the segment flag indicating where the *path* parameter is located. The only valid value is: **FP_SYS** The *path* parameter is stored in kernel memory.
countp	Points to the location where a count of bytes actually written is to be returned (must be in kernel space). GDLC does not provide this information for a kernel user since **mbufs** are used, but the file system requires a valid address and writes a copy of the *nbytes* parameter to that location.

Description

Four types of data can be sent to generic data link control (GDLC). Network data can be sent to a service access point (SAP), and normal, exchange identification (XID) or datagram data can be sent to a link station (LS).

Kernel users pass a communications memory buffer (**mbuf**) directly to GDLC on the **fp_write** kernel service. In this case, a **uiomove** kernel service is not required, and maximum performance can be achieved by merely passing the buffer pointer to GDLC. Each write buffer is required to have the proper buffer header information and enough space for the data link headers to be inserted. A write data offset is passed back to the kernel user at start LS completion for this purpose.

All data must fit into a single packet for each write call. That is, GDLC does not separate the user's write data area into multiple transmit packets. A maximum write data size is passed back to the user at **DLC_ENABLE_SAP** completion and at **DLC_START_LS** completion for this purpose.

Normally, a write subroutine can be satisfied immediately by GDLC by completing the data link headers and sending the transmit packet down to the device handler. In some cases, however, transmit packets can be blocked by the particular protocol's flow control or a resource outage. GDLC reacts to this differently, based on the system blocked/nonblocked file status flags (set by the file system and based on the **O_NDELAY** and **O_NONBLOCKED** values passed on the **fp_open** kernel service). Nonblocked **write** subroutines that cannot get enough resources to queue the communications memory buffer (**mbuf**) return an error indication. Blocked write subroutines put the calling process to sleep until the resources free up or an error occurs. Each GDLC supports the **fp_write** kernel service via its **dlcwrite** entry point. The **fp_write** kernel service may be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EAGAIN	Indicates that transmit is temporarily blocked, and the calling process cannot be put to sleep.

Item	Description
EINTR	Indicates that a signal interrupted the kernel service before it could complete successfully.
EINVAL	Indicates an invalid argument, such as too much data for a single packet.
ENXIO	Indicates an invalid file pointer.

These return values are defined in the **/usr/include/sys/errno.h** file.

Related reference:

"fp_write Kernel Service" on page 171

Related information:

Generic Data Link Control (GDLC) Environment Overview

Parameter Blocks by ioctl Operation for DLC

fp_writev Kernel Service

Purpose

Performs a write operation on an open file with arguments passed in **iovec** elements.

Syntax

#include <sys/types.h> #include <sys/errno.h> int fp_writev (*fp*, *iov*, *iovcnt*, *ext*, *seg*, *countp***) struct file *** *fp***; struct iovec *** *iov***; ssize_t** *iovcnt***; ext_t** *ext***; int** *seg***; ssize_t *** *countp***;**

Parameters

Item	Description
fp	Points to a file structure returned by the **fp_open** kernel service.
iov	Points to an array of **iovec** elements. Each **iovec** element describes a buffer containing data to be written to the file.
iovcnt	Specifies the number of **iovec** elements in an array pointed to by the *iov* parameter.
ext	Specifies an extension argument required by some device drivers. Its content, form, and use are determined by the individual driver.
segflag	Indicates which part of memory the information designated by the *iov* parameter is located in: **SYS_ADSPACE** The information designated by the *iov* parameter is in kernel memory. **USER_ADSPACE** The information designated by the *iov* parameter is in application memory.
countp	Points to the location where the count of bytes actually written to the file is to be returned.

Description

The **fp_writev** kernel service is an internal interface to the function provided by the **writev** subroutine.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.

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

Related information:

writev subroutine

Logical File System Kernel Services

fskv_reg Kernel Service

Purpose

Registers callout handlers for validation of file system operations.

Syntax

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

int fskv_reg(fskv_t *fs_kv, ulong options);
```

```
typedef struct fskv {
       int version_number;
       int (*kv_open)(struct xfid *xfp,
                           long flags,
                           cred_ext_t *crxp);
       int (*kv_setattr)(struct xfid *xfp,
                           long op,
                           long arg1,
                           long arg2,
                           long arg3,
                           cred_ext_t *crxp);
} fskv_t;
```

Parameters

fs_kv
Specifies an array of callout functions that are called to validate file system operations in the kernel.

options
Specifies a bit mask of registration options. The **options** parameter is not defined currently. The caller must set the **options** parameter to 0.

Description

The fskv_reg kernel service registers an array of functions that are called before the execution of file system-specific operations.

After a callout handler is registered, each of the affected operations is preceded by a call to the corresponding validation routine.

Only one callout array can be registered. After a callout array is registered with the fskv_reg kernel service, the subsequent invocation of the fskv_reg kernel service does not succeed until the fskv_unreg kernel service is called. The caller of the fskv_reg kernel service must have root authority.

Execution environment

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

Return values

On successful completion, the fskv_reg kernel service returns a value of 0.

The following error codes can be returned on failure:

EEXIST
The callout array is already registered.

EPERM
The caller does not have permission to invoke this function.

EINVAL
A parameter is invalid.

Callout handlers

Callouts can be specified for the open, chmod, chown, and utimes system calls. The chmod, chown, and utimes system calls are handled in a single operation in the kernel with the setattr call.

If the validation callout routine returns a nonzero value, the file system operation is stopped and the system call returns the EPERM value.

The validation routines are called only for local physical file systems (JFS2 and JFS) and network file system (NFS)-mounted file systems. The callout functions accept the following arguments. The `xfid` argument uniquely identifies the file within the current running system.

```
typedef struct xfid {
       fsid_t x_fsid;
       fid_t x_fid;
} xfid_t;
```

kv_open() callout function

The `kv_open` callout function contains the information that is available to the open routines of the file system to track and validate open calls.

Syntax

```
#include <sys/file.h>
#include <sys/cred.h>

int    (*kv_open)(struct xfid *xfp,
          long flags,
          void *nrp,
          cred_ext_t *crxp);
```

Parameters

xfp
Pointer to an `xfid` structure that identifies the file system and object.

nrp
Name resolution information. If the `xfidToName` kernel service is called, this parameter must be passed to it. This pointer is not valid after it is returned from the callout function.

flags
Open flags that are passed by the application.

crxp
Pointer to the credentials for the calling process.

Return values

Zero
Indicates that the validation completed successfully.

Nonzero
Indicates that the validation failed.

kv_setattr() callout function

The `kv_setattr` callout function contains the information that is available to the system call that initiated this function. The `setattr` function is called by the `chown`, `chmod`, and `utimes` system calls and the variants of those system calls (for example, `fchown`, `fchmod`, and `futimens` system calls).

Syntax

```
#include <sys/vattr.h>
#include <sys/cred.h>

int (*kv_setattr)(struct xfid *xfp,
          long op,
          long arg1,
          long arg2,
          long arg3,
          void *nrp,
          cred_ext_t *crxp);
```

Parameters

xfp
Pointer to an `xfid` structure that identifies the file system and object.

op Specifies one of the following operations:

V_OWN
Sets file ownership.

V_MODE
Sets file mode.

V_UTIME
Sets the file time specified by the user.

V_STIME
Sets the file time requested by the system.

arg*n*
Specifies the following values for each of the listed operations.

Table 1. kv_setattr() callout function: argn parameter values

Operations	arg1	arg2	arg3

V_OWN	flag:
	`T_OWNER_AS_IS`
	`T_GROUP_AS_IS`
	(For information about the file ownership changes, see chownx subroutine.)
		`uid_t newuid`	`gid_t newgid`

V_MODE	`mode_t newmode`	Unused	Unused
V_UTIME	flag: `V_SETTIME` Ignore arguments and set time to the current time.	`timestruct_t *atime` Set the access time.	`timestruct_t *mtime` Set the modification time.
V_STIME	`NULL` or `timestruct_t *atime` Set the access time.	`NULL` or `timestruct_t *mtime` Set the modification time.	`NULL` or `timestruct_t *ctime` Set the change time.

nrp
Indicates the name resolution information. If the `xfidToName` kernel service is called, this parameter must be passed to it. This parameter is a pointer to temporary information that is not valid after it is returned from the validation routine.

crxp
Pointer to credentials for the calling process.

Return values

Zero
Indicates that the validation completed successfully.

Nonzero
Indicates that the validation failed.

Related reference:
"nameToXfid() Kernel Service" on page 376
"xfidToName() Kernel Service" on page 590

fskv_unreg Kernel Service

Purpose

Unregisters callout handlers for validation of file system operations.

Syntax

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

 int fskv_unreg(ulong options);
```

Parameters

`options`
Specifies a bit mask of registration options. The **`options`** parameter is not defined currently. The caller must set the **`options`** parameter to `0`.

Description

The `fskv_unreg` kernel service removes the registration of the functions to be called before the execution of file system operations. After this service completes, the callout handlers are not executed.

The caller of the `fskv_unreg` kernel service must have root authority.

Execution environment

The `fskv_unreg` kernel service can be called only from the process environment.

Return values

On successful completion, the `fskv_unreg` kernel service returns a value of 0.

The following error codes are returned on failure:

EPERM
The caller does not have permission to invoke this function.

EINVAL
A parameter is invalid.

fubyte Kernel Service

Purpose

Retrieves a byte of data from user memory.

Syntax

#include <sys/types.h> #include <sys/errno.h> int fubyte (*uaddr***) uchar** **uaddr*;

Parameter

Item	Description
uaddr	Specifies the address of the user data.

Description

The **fubyte** kernel service fetches, or retrieves, a byte of data from the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **fubyte** service ensures that the user has the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

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

Execution Environment

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

Return Values

When successful, the **fubyte** service returns the specified byte.

Item	Description
-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.

Related reference:

"fuword Kernel Service"

"subyte Kernel Service" on page 477

Related information:

Accessing User-Mode Data while in Kernel Mode

fuword Kernel Service

Purpose

Retrieves a word of data from user memory.

Syntax

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

int fuword ( uaddr)
int *uaddr;
```

Parameter

Item	Description
uaddr	Specifies the address of user data.

Description

The **fuword** kernel service retrieves a word of data from the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **fuword** service ensures that the user had the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

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

Execution Environment

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

Return Values

When successful, the **fuword** service returns the specified word of data.

Item	Description
-1	Indicates a *uaddr* parameter that is not valid.

The access is not valid under the following circumstances:

- The user does not have sufficient authority to access the data.
- The address is not valid.
- An I/O error occurred while referencing the user data.

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

Related reference:

“fubyte Kernel Service” on page 178

“subyte Kernel Service” on page 477

Related information:

Accessing User-Mode Data while in Kernel Mode

g

The following kernel services begin with the with the letter g.

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

Item	Description
dev	Specifies the device that contains the block to be allocated.
blkno	Specifies the block to be allocated.

Description

The **getblk** kernel service first checks whether the specified buffer is in the buffer cache. If the buffer resides there, but is in use, the **e_sleep** service is called to wait until the buffer is no longer in use. Upon waking, the **getblk** service tries again to access the buffer. If the buffer is in the cache and not in use, it is removed from the free list and marked as busy. Its buffer header is then returned. If the buffer is not in the buffer cache, another buffer is taken from the free list and returned.

Execution Environment

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

Return Values

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

On a platform that supports storage keys, the buffer header is allocated from the storage that is protected by the **KKEY_BLOCK_DEV** kernel key.

Related reference:

"bread Kernel Service" on page 30

Related information:

Block I/O Buffer Cache Kernel Services: Overview

I/O Kernel Services

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

Item	Description
header	Specifies the address of the **clist** structure that describes the character list.

Description

> **Attention:** The caller of the **getc** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Otherwise, the system may crash.

The **getc** kernel service returns the character at the front of the character list. After returning the last character in the buffer, the **getc** service frees that buffer.

Execution Environment

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

Return Values

Item	Description
-1	Indicates that the character list is empty.

Related information:

I/O Kernel Services

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

Item	Description
header	Specifies the address of the **clist** structure that describes the character list.

Description

> **Attention:** The caller of the **getcb** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character buffers acquired from the **getcf** service are pinned. Otherwise, the system may crash.

The **getcb** kernel service returns the address of the character buffer at the start of the character list and removes that buffer from the character list. The user must free the buffer with the **putcf** service when finished with it.

Execution Environment

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

Return Values

A null address indicates the character list is empty.

The **getcb** service returns the address of the character buffer at the start of the character list when the character list is not empty.

Related reference:

"getcf Kernel Service" on page 184

Related information:

I/O Kernel Services

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

Item	Description
header	Specifies the address of the **clist** structure that describes the character list.
dest	Specifies the address where the characters obtained from the character list are to be placed.
n	Specifies the number of characters to be read from the character list.

Description

> **Attention:** The caller of the **getcbp** services must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character buffers acquired from the **getcf** service are pinned. Otherwise, the system may crash.

The **getcbp** kernel service retrieves as many as possible of the *n* characters requested from the character buffer at the start of the character list. The **getcbp** service then places them at the address pointed to by the *dest* parameter.

Execution Environment

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

Return Values

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

Related reference:

"getcf Kernel Service" on page 184

Related information:

I/O Kernel Services

getcf Kernel Service

Purpose

Retrieves a free character buffer.

Syntax

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

```
struct cblock *getcf ( )
```

Description

The **getcf** kernel service retrieves a character buffer from the list of available ones and returns that buffer's address. The returned character buffer is pinned. If you use the **getcf** service to get a character buffer, be sure to free the space when you have finished using it. The buffers received from the **getcf** service should be freed by using the **putcf** kernel service.

Before starting the **getcf** service, the caller should request enough **clist** resources by using the **pincf** kernel service. The proper use of the **getcf** service ensures that there are sufficient pinned buffers available to the caller.

If the **getcf** service indicates that there is no available character buffer, the **waitcfree** service can be called to wait until a character buffer becomes available.

The **getcf** service has no parameters.

Execution Environment

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

Return Values

Upon successful completion, the **getcf** service returns the address of the allocated character buffer.

A null pointer indicates no buffers are available.

Related reference:

"pincf Kernel Service" on page 409

"putcf Kernel Service" on page 425

Related information:

I/O Kernel Services

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

Item	Description
header	Specifies the address of the **clist** structure that describes the character list.

Description

> **Attention:** The caller of the **getcx** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character buffers acquired from the **getcf** service are pinned.

The **getcx** kernel service is identical to the **getc** service, except that the **getcx** service returns the character at the end of the list instead of the character at the front of the list. The character at the end of the list is the last character in the first buffer, not in the last buffer.

Execution Environment

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

Return Values

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

Related reference:

"getcf Kernel Service" on page 184

Related information:

I/O Kernel Services

geteblk Kernel Service

Purpose

Allocates a free buffer.

Syntax

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

struct buf *geteblk ( )
```

Description

Attention: The use of the **geteblk** service by character device drivers is strongly discouraged. As an alternative, character device drivers can use the **xmalloc** service to allocate the memory space directly, or the character I/O kernel services such as the **getcb** or **getcf** services.

The **geteblk** kernel service allocates a buffer and buffer header and returns the address of the buffer header. If no free buffers are available, then the **geteblk** service waits for one to become available. Block device drivers can retrieve buffers using the **geteblk** service.

In the header, the `b_forw`, `b_back`, `b_flags`, `b_bcount`, `b_dev`, and `b_un` fields are used by the system and cannot be modified by the driver. The `av_forw` and `av_back` fields are available to the user of the **geteblk** service for keeping a chain of buffers by the user of the **geteblk** service. (This user could be the kernel file system or a device driver.) The `b_blkno` and `b_resid` fields can be used for any purpose.

The **brelse** service is used to free this type of buffer.

The **geteblk** service has no parameters.

Execution Environment

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

Return Values

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

On a platform that supports storage keys, the buffer header is allocated from the storage protected by the **KKEY_BLOCK_DEV** kernel key.

Related reference:

"xmalloc Kernel Service" on page 597

"buf Structure" on page 614

Related information:

Block I/O Buffer Cache Kernel Services: Overview

geteror 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

Item	Description
bp	Specifies the address of the buffer structure whose status is to be checked. On a platform that supports storage keys, the passed in *bp* parameter must be in the **KKEY_PUBLIC** or **KKEY_BLOCK_DEV** protection domain.

Description

The **geterror** kernel service checks the specified buffer to see if the **b_error** flag is set. If that flag is not set, the **geterror** service returns 0. Otherwise, it returns the nonzero **B_ERROR** value or the **EIO** value (if **b_error** is 0).

Execution Environment

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

Return Values

Item	Description
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.

Related information:

Block I/O Buffer Cache Kernel Services: Overview

I/O Kernel Services

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

Item	Description
exceptp	Specifies the address of an **except** structure, as defined in the **/usr/include/sys/except.h** file. The **getexcept** service copies detailed exception data from the current machine-state save area into this caller-supplied structure.

Description

The **getexcept** kernel service provides exception handlers the capability to retrieve additional information concerning the exception from the machine-state save area.

The **getexcept** service should only be used by exception handlers when called to handle an exception. The contents of the structure pointed at by the *exceptp* parameter is platform-specific, but is described in the **/usr/include/sys/except.h** file for each type of exception that provides additional data. This data is typically included in any error logging data for the exception. It can be also used to attempt to handle or recover from the exception.

Execution Environment

The **getexcept** kernel service can be called from either the process or interrupt environment. It should be called only when handling an exception.

Return Values

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

Related information:

Kernel Extension and Device Driver Management Kernel Services and

getfslimit Kernel Service

Purpose

Returns the maximum file size limit of the current process.

Syntax

#include <sys/types.h> offset_t getfslimit (void)

Description

The **getfslimit** kernel service returns the file size limit of the current process as a 64 bit integer. This can be used by file systems to implement the checks needed to enforce limits. The **getfslimit** kernel service is called from the process environment.

Return Values

The **getfslimit** kernel service returns the the file size limit, there are no error values.

Related information:

ulimit subroutine

getrlimit subroutine

ulimit subroutine

get_pag or get_pag64 Kernel Service

Purpose

Retrieves a Process Authentication Group (PAG) value for the current process.

Syntax

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

int get_pag ( type, pag )
int type;
int *pag;

int get_pag64 ( type, pag )
int type;
uint64_t *pag;
```

Parameters

Item	Description
type	PAG type to retrieve
pag	Pointer to buffer where operating system returns the PAG

Description

The **get_pag** and **get_pag64** kernel services copy the requested PAG from the current process into *pag*. The value of *type* must be a defined PAG ID. The PAG ID for the Distributed Computing Environment (DCE) is 0.

Execution Environment

The **get_pag** and **get_pag64** kernel services can be called from the process environment only.

Return Values

A value of `0` is returned upon successful completion. If unsuccessful, **errno** is set to a value that explains the error.

Error Codes

The **get_pag** kernel service fails if one or both of the following conditions are true:

Item	Description
EINVAL	Invalid PAG specification
EOVERFLOW	PAG value is 64-bit (should be using **get_pag64**)

The **get_pag64** kernel service fails if the following condition is true:

Item	Description
EINVAL	Invalid PAG specification

Related information:

Security Kernel Services

getpid Kernel Service

Purpose

Gets the process ID of the current process.

Syntax

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

```
pid_t getpid ()
```

Description

The **getpid** kernel service returns the process ID of the calling process.

The **getpid** service can also be used to check the environment that the routine is being executed in. If the caller is executing in the interrupt environment, the **getpid** service returns a process ID of -1. If a routine is executing in a process environment, the **getpid** service obtains the current process ID.

Execution Environment

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

Return Values

Item	Description
-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.

Related information:

Process and Exception Management Kernel Services

Understanding Execution Environments

getppidx Kernel Service

Purpose

Gets the parent process ID of the specified process.

Syntax

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

pid_t getppidx (ProcessID)
pid_t ProcessID;
```

Parameter

Item	Description
ProcessID	Specifies the process ID. If this parameter is 0, then the parent process ID of the calling process is returned.

Description

The **getppidx()** kernel service accepts a process ID as an input. If the input process ID is 0, the **getppidx()** subroutine returns the process ID of the calling process' parent process. If the input process ID is nonzero and a valid value, the parent ID of the input process ID is returned. If the input process ID is invalid, the **getppidx()** kernel service returns -1.

Execution Environment

The **getppidx()** kernel service can be called from the process environment only.

Return Values

Item	Description
-1	Indicates that the **ProcessID** parameter is invalid.

Related reference:

"getpid Kernel Service" on page 189

Related information:

Process and Exception Management Kernel Services

Understanding Execution Environments

getuerror Kernel Service

Purpose

Allows kernel extensions to read the **ut_error** field for the current thread.

Syntax

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

int getuerror ()
```

Description

The **getuerror** kernel service allows a kernel extension in a process environment to retrieve the current value of the current thread's **ut_error** field. Kernel extensions can use the **getuerror** service when using system calls or other kernel services that return error information in the **ut_error** field.

For system calls, the system call handler copies the value of the **ut_error** field in the per thread **uthread** structure to the **errno** global variable before returning to the caller. However, when kernel services use available system calls, the system call handler is bypassed. The **getuerror** service must then be used to obtain error information.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.

When an error occurs, the **getuerror** kernel service returns the current value of the **ut_error** field in the per thread **uthread** structure. Possible return values for this field are defined in the **/usr/include/sys/errno.h** file.

Related reference:

"setuerror Kernel Service" on page 468

Related information:

Kernel Extension and Device Driver Management Kernel Services

Understanding System Call Execution

getufdflags and setufdflags Kernel Services

Purpose

Queries and sets file-descriptor flags.

Syntax

#include <sys/user.h> int getufdflags(*fd, flagsp***) int** *fd***; int** **flagsp*; **#include <sys/user.h> int setufdflags(***fd*, *flags*) **int** *fd*; **int** *flags***;**

Parameters

Item	Description
fd	Identifies the file descriptor.
flags	Sets attribute flags for the specified file descriptor. Refer to the **sys/user.h** file for the list of valid flags.
flagsp	Points to an integer field where the flags associated with the file descriptor are stored on successful return.

Description

The **setufdflags** and **getufdflags** kernel services set and query the file descriptor flags. The file descriptor flags are listed in **fontl.h**.

Execution Environment

These kernel services can be called from the process environment only.

Return Values

Item	Description
0	Indicates successful completion.
EBADF	Indicates that the *fd* parameter is not a file descriptor for an open file.

Related reference:

"ufdhold and ufdrele Kernel Service" on page 515

get_umask Kernel Service

Purpose

Queries the file mode creation mask.

Syntax

```
int get_umask(void)
```

Description

The **get_umask** service gets the value of the file mode creation mask currently set for the process.

Note: There is no corresponding kernel service to set the umask because kernel routines that need to set the umask can call the **umask** subroutine.

Execution Environment

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

Return Values

The **get_umask** kernel service always completes successfully. Its return value is the current value of the **umask**.

Related information:

umask subroutine

gfsadd Kernel Service

Purpose

Adds a file system type to the **gfs** table.

Syntax

#include <sys/types.h> #include <sys/errno.h> int gfsadd (*gfsno*, *gfsp***) int** *gfsno***; struct gfs ****gfsp***;**

Parameters

Item	Description
gfsno	Specifies the file system number. This small integer value is either defined in the **/usr/include/sys/vmount.h** file or a user-defined number of the same order.
gfsp	Points to the file system description structure.

Description

The **gfsadd** kernel service is used during configuration of a file system. The configuration routine for a file system invokes the **gfsadd** kernel service with a **gfs** structure. This structure describes the file system type.

The **gfs** structure type is defined in the **/usr/include/sys/gfs.h** file. The **gfs** structure must have the following fields filled in:

Field **Description**

`gfs_type`
Specifies the integer type value. The predefined types are listed in the **/usr/include/sys/vmount.h** file.

`gfs_name`
Specifies the character string name of the file system. The maximum length of this field is 16 bytes. Shorter names must be null-padded.

`gfs_flags`
Specifies the flags that define the capabilities of the file system. The following flag values are defined:

GFS_AHAFS_INFO
GFS supports AHAFS FS monitoring.

GFS_AIX_FLOCK
Uses **common_reclock()** to manage advisory locks.

GFS_DIROP
Call parent **vnop** instead of **obj**.

GFS_FASTPATH
GFS supports AIO fast path.

GFS_FUMNT
File system supports forced unmount.

GFS_INIT
GFS has been initialized

GFS_MEMCNTL
New **memcntl** vnode operation

GFS_MLS
GFS supports MLS.

GFS_NAMED_OPEN
File system supports named open.

GFS_NO_ACCT
Do not do file system account on this file system.

GFS_NOEXPORT
GFS cannot be exported by NFS.

GFS_NOUMASK
File system does not apply umask when creating new objects.

GFS_OFLAGS64
GFS supports 64 bit open flags.

GFS_REMNT
File system supports remount of a mounted file system.

Field **Description**

GFS_REMOTE
File system is remote (ie. NFS).

GFS_STATFSVP
File system supports **vfs_statfsvp** VFS interface. (new vfs operation: **vfs_statfsvp**)

GFS_SYS5DIR
File system that uses the System V-type directory structure.

GFS_SYNCVFS
The **syncvfs** vnode operation.

GFS_VERSION4
File system supports AIX Version 4 V-node interface.

GFS_VERSION42
File system supports AIX 4.2 V-node interface. (new vnode operation: vnop_seek)

GFS_VERSION421
File system supports AIX 4.2.1 V-node interface.(new vnode operations: **vnop_sync_range**, **vnop_create_attr**, **vnop_finfo**, **vnop_map_lloff**, **vnop_readdir_eofp**, **vnop_rdwr_attr**)

GFS_VERSION43
File system supports AIX 4.3 V-node interface. (new file flag for vnop_sync_range:FMSYNC)

GFS_VERSION53
File system supports AIX 5.3 V-node interface (new vnode operations: vnop_getxacl, vnop_setxacl) and AIX 5.3 VFS interface. (new vfs operation: vfs_aclxcntl)

GFS_VREGSEL
GFS wants to select vnode operation called for **VREG** files.

`gfs_ops` Specifies the array of pointers to **vfs** operation implementations.

`gn_ops` Specifies the array of pointers to v-node operation implementations.

The file system description structure can also specify:

Item	Description
`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

Item	Description
0	Indicates successful completion.
EBUSY	Indicates that the file system type has already been installed.
EINVAL	Indicates that the *gfsno* value is larger than the system-defined maximum. The system-defined maximum is indicated in the **/usr/include/sys/vmount.h** file.

Related reference:

“gfsdel Kernel Service”

“vfs_init Entry Point” on page 644

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

Item	Description
gfsno	Specifies the file system number. This value identifies the type of the file system to be deleted.

Description

The **gfsdel** kernel service is called to delete a file system type. It is not valid to mount any file system of the given type after that type has been deleted.

Execution Environment

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

Return Values

Item	Description
0	Indicates successful completion.
ENOENT	Indicates that the indicated file system type was not installed.
EINVAL	Indicates that the *gfsno* value is larger than the system-defined maximum. The system-defined maximum is indicated in the **/usr/include/sys/vmount.h** file.
EBUSY	Indicates that there are active **vfs** structures for the file system type being deleted.

Related reference:

"gfsadd Kernel Service" on page 192

Related information:

Virtual File System Overview

Virtual File System Kernel Services

gn_closecnt Subroutine

Purpose

Maintains the using count on a **gnode** structure.

Syntax

```
#include <sys/vnode.h>
#include <sys/fcntl.h>

void gn_closecnt (gnode, flags)
struct gnode *gnode;
long flags;
```

Parameters

Item	Description
gnode	Points to a **gnode** structure.
flags	Specifies the open mode (**FREAD**, **FWRITE**, **FEXEC**, **FRSHARE**) from the open file flags.

Description

The **gn_closecnt** subroutine uses the passed in *flags* value to determine the appropriate using counts to decrease in the *gnode* structure. For example, if the **FREAD** flag is set, the **gn_closecnt** subroutine decreases the gn_rdcnt field. The following table shows the mapping of the *flags* value to the counts field in the *gnode* structure:

Item	Description
FREAD	gn_rdcnt
FWRITE	gn_wrcnt
FEXEC	gn_excnt
FRSHARE	gn_rshcnt

Return Values

The **gn_closecnt** subroutine returns no return values.

Error Codes

The **gn_closecnt** subroutine returns no error codes.

Related information:

Understanding Data Structures and Header Files for Virtual File Systems

gn_common_memcntl Subroutine

Purpose

Changes or queries the physical attachment of a file.

Syntax

```
#include <sys/vnode.h>
#include <sys/fcntl.h>

int gn_common_memcntl (gnode, cmd, arg)
struct gnode * gnode;
int cmd;
void * arg;
```

Parameters

Item	Description
gnode	Points to a **gnode** structure.
cmd	Specifies the operation to be performed. The *cmd* parameter can be one of the following values: • **F_ATTACH** • **F_DETACH** • **F_ATTINFO**
arg	Points to a structure containing information for the specified *cmd* parameter. **F_ATTACH** attach_desc_t **F_DETACH** detach_desc_t **F_ATTINFO** attinfo_desc_t

Description

The **gn_common_memcntl** subroutine is to be called by file system **vnop_memcntl** implementations. It performs the normal function of such operations. If the *cmd* parameter is set to **F_ATTACH**, the **gn_common_memcntl** subroutine attaches the segment specified by the *gn_seg* field in the **gnode** structure. If the *cmd* parameter is set to **F_DETACH**, the **gn_common_memcntl** subroutine detaches the segment. If the *cmd* parameter is set to **F_ATTINFO**, the **gn_common_memcntl** subroutine returns information about the current state of attachment.

Return Values

Item	Description
0	Success.
non-zero	Failure.

Error Codes

Item	Description
EINVAL	The *cmd* parameter is not valid.
ENOMEM	Resources are not available to attach the memory segment.

gn_mapcnt Subroutine

Purpose

Maintains the mapping count in a **gnode** structure.

Syntax

```
#include <sys/vnode.h>
#include <sys/shm.h>

void gn_mapcnt (gnode, flags)
struct gnode * gnode;
long flags;
```

Parameters

Item	Description
gnode	Points to a **gnode** structure.
flags	Specifies the following mapping flag: **SHM_RDONLY** Only read access is required.

Description

The **gn_mapcnt** subroutine uses the passed in *flags* value to determine the appropriate mapping count to increase in the *gnode* structure. If the **SHM_RDONLY** flag is set, the **gn_mapcnt** subroutine increases the gn_mrdcnt field. Otherwise, the **gn_mapcnt** subroutine increases the gn_mwrcnt field.

Return Values

The **gn_mapcnt** subroutine returns no return values.

Error Codes

The **gn_mapcnt** subroutine returns no error codes.

Related information:

mmap subroutine

shmat subroutine

gn_opencnt Subroutine

Purpose

Maintains the using count on a **gnode** structure.

Syntax

```
#include <sys/vnode.h>
#include <sys/fcntl.h>

void gn_opencnt (gnode, flags)
struct gnode * gnode;
long flags;
```

Parameters

Item	Description
gnode	Points to a **gnode** structure.
flags	Specifies the open mode (**FREAD**, **FWRITE**, **FEXEC**, **FRSHARE**) from the open file flags.

Description

The **gn_opencnt** subroutine uses the passed in *flags* value to determine the appropriate using counts to increase in the *gnode* structure. The following table shows the mapping of the *flags* value to the counts field in the *gnode* structure:

Item	Description
FREAD	gn_rdcnt
FWRITE	gn_wrcnt
FEXEC	gn_excnt
FRSHARE	gn_rshcnt

Return Values

The **gn_opencnt** subroutine returns no return values.

Error Codes

The **gn_opencnt** subroutine returns no error codes.

Related information:

Understanding Data Structures and Header Files for Virtual File Systems

gn_unmapcnt Subroutine

Purpose

Maintains the mapping count in a **gnode** structure.

Syntax

```
#include <sys/vnode.h>
#include <sys/shm.h>

void gn_unmapcnt (gnode, flags)
struct gnode * gnode;
long flags;
```

Parameters

Item	Description
gnode	Points to a **gnode** structure.
flags	Specifies the following mapping flag: **SHM_RDONLY** Only read access is required.

Description

The **gn_unmapcnt** subroutine uses the passed in *flags* value to determine the appropriate mapping count to decrease in the *gnode* structure. If the **SHM_RDONLY** flag is set, the **gn_unmapcnt** subroutine decreases the gn_mrdcnt field. Otherwise, the **gn_unmapcnt** subroutine decreases the gn_mwrcnt field.

Return Values

The **gn_unmapcnt** subroutine returns no return values.

Error Codes

The **gn_unmapcnt** subroutine returns no error codes.

Related information:

mmap subroutine

shmat subroutine

groupmember, groupmember_cr Subroutines

Purpose

Determines if the named group is a member of a credential group set.

Syntax

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

int groupmember (gid)
gid_t gid;

int groupmember_cr (gid, cred)
gid_t gid;
cred_t * cred;
```

Parameters

Item	Description
gid	Specifies an identifier for a group.
cred	Points to a **ucred** structure.

Description

The **groupmember** subroutines determine if a group is included in the group set of a credential structure. The **groupmember** subroutine queries the credential associated with the current thread. The **groupmember_cr** subroutine checks for the group within the specified **ucred** structure.

Return Values

The **groupmember** subroutines return TRUE if the **ucred** structure contains the specified *gid* parameter or if the specified *gid* parameter is the current effective group ID for the thread. Otherwise, these routines return FALSE.

Error Codes

The **groupmember** subroutines return no error codes.

Related information:

Security Kernel Services

h

The following kernel services begin with the with the letter h.

heap_create Kernel Service

Purpose

Initializes a new heap to be used with kernel memory management services. The **heap_create** kernel service replaces the **init_heap** kernel service. It returns a heap handle that can be used with the **xmalloc** and the **xmfree** kernel services.

Syntax

```
#include <sys/types.h>
#include <sys/malloc.h>
#include <sys/skeys.h>
#include <sys/kerrno.h>

kerrno_t heap_create (heapattr_t * heapattr, heapaddr_t * heapptr);
```

Parameters

Item	Description
heapattr	Points to an initialized heap attribute structure. See the **sys/malloc.h** file. This structure is initialized by the caller of **heap_create**.
heapptr	Points to an external heap descriptor. The caller must initialize this parameter to the **HPA_INVALID_HEAP** value.

The *heapattr* structure contains the following fields:

Item	Description
eye_catch8b_t hpa_eyec	Must be initialized to the **EYEC_HEAPATTR** value.
short hpa_version	Must be initialized to the **HPA_VERSION** value.
long hpa_flags	The following flags describe heap properties: **HPA_PAGED** The heap returns pageable memory. **HPA_PINNED** The heap returns pinned memory. **HPA_SHARED** The returned descriptor is backed by a common sub-heap. **HPA_PRIVATE** The returned descriptor is backed by isolated storage.
*void * hpa_heapaddr*	Must be set to NULL (reserved).
size_t hpa_heapsize	Heap size in bytes. It is only used for private heaps.
size_t hpa_limit	Usage barrier independent from size. Limits the amount available from a private heap that is less than or equal to the actual size of the private heap.
long hpa_debug_level	Heap debug level. The **HPA_DEFAULT_DEBUG** value gives the heap the system debug level.
uint hpa_kkey	Kernel key requested for the storage allocated.

Description

The **heap_create** service is a replacement for the **init_heap** service. It can be used to create private heaps, and to create shared sub-heaps. After this service creates a private heap or a handle to a shared sub-heap, the returned **heapaddr_t** value can be used with the **xmalloc** service or the **xmfree** service to allocate or free memory from that heap.

The most common usage for the **heap_create** service is to get a handle to a shared sub-heap. This is done by setting the **HPA_SHARED** flag in the input attribute structure. See the **sys_malloc.h** file.

Private heaps can be created by specifying the **HPA_PRIVATE** flag. This allows the **heap_create** service to initialize and manage an area of virtual memory as a private heap. The **hpa_heapaddr** field must be set to zero. The **heap_create** service provides the storage but this field is reserved for future use. The **hpa_size** field indicates the size of the private heap in bytes.

Private heaps can make use of the **hpa_limit** field. Use the **hpa_size** field to reserve a maximum effective address space. Then use the **hpa_limit** field to alter and control the amount of effective address space that is in use. The value of the **hpa_limit** field must be less than or equal to the value of the **hpa_size** field.

The **hpa_debug** and **hpa_kkey** fields are required for shared and private heaps. The **hpa_debug** level allows a component run-time debug level to be applied to allocations using the returned heap handle. The **hpa_kkey** field associates a kernel key with a sub-heap that can limit the kernel accessibility.

On a successful completion, the *heapattr* field contains the address of a heap structure. This can be used as a parameter to the **xmalloc** and the **xmfree** kernel services. The memory returned by these services and the internal heap structures are protected by the **hpa_kkey** field. When calling the **xmalloc** and the **xmfree** heap services, the caller must hold the key that was used when creating the heap.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion. A descriptor is returned in the *heapptr* parameter.
EINVAL_HEAP_CREATE	Indicates one or more of the following inputs that were not valid: • heapattr is NULL. • *heapptr != HPA_INVALID_HEAP. • heapattr->hpa_eyec != EYEC_HEAPATTR. • heapattr->hpa_version != HPA_VERSION. • Flags: Both the **HPA_SHARED** and the **HPA_PRIVATE** flags are specified. • Flags: Neither the **HPA_SHARED** nor the **HPA_PRIVATE** flag is specified. • Flags: Both the **HPA_PINNED** and the **HPA_PAGED** flags are specified. • Flags: Neither the **HPA_PINNED** nor the **HPA_PAGED** flag is specified. • Keys: kernel key specified is not valid. • Other: Size is specified with a shared heap. • Other: Limit is specified with a shared heap. • Other: Address specified is not NULL. • Other: Limit > size for private heap. • Other: Private heap size is too small (less than 8M).
ENOMEM_HEAP_CREATE	Indicates insufficient memory available to complete the request.

Related reference:

"heap_modify Kernel Service" on page 203

"heap_destroy Kernel Service"

heap_destroy Kernel Service

Purpose

Removes a heap.

Syntax

```
#include <sys/types.h>
#include <sys/malloc.h>
#include <sys/kerrno.h>

kerrno_t heap_destroy (heapattr_t heap, long flags);
```

Parameters

Item	Description
heap	The heap to destroy.
flags	Must be zero.

Description

This service removes a heap and its internal resources from the system. There must be no outstanding allocations when a heap is destroyed.

Execution Environment

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

Return Values

Item	Description
EINVAL_HEAP_DESTROY	The *heap* parameter is not recognizable.
EBUSY_HEAP_DESTROY	The heap is still in use.

Related reference:

"heap_create Kernel Service" on page 200

"heap_modify Kernel Service"

heap_modify Kernel Service

Purpose

Modifies the attributes of a heap.

Syntax

```
#include <sys/types.h>
#include <sys/malloc.h>
#include <sys/kerrno.h>

kerrno_t heap_modify (heapattr_t heap, long command, long argument);
```

Parameters

Item	Description
heap	The heap handle returned from the **heap_create** kernel service.
command	Specifies the operation to perform. The following values are supported: **HPA_SET_LIMIT** Modifies the limit value of a private heap. **HPA_SET_DEBUG** Modifies the debug level. Debug levels from 0 to 9 are supported.
argument	Command specific data (new limit or debug level).

Description

The **heap_modify** kernel service is used to alter the heap characteristics at run time.

Execution Environment

The **heap_modify** kernel service can be called from the process environment only with interrupts enabled.

Return Values

Item	Description
0	Success.
EINVAL_HEAP_MODIFY	The command or the execution environment is not valid.
ERANGE_HEAP_MODIFY	Heap property is outside the supported range.

Related reference:

"heap_create Kernel Service" on page 200

"heap_destroy Kernel Service" on page 202

hkeyset_add, hkeyset_replace, hkeyset_restore, or hkeyset_get Kernel Service

Purpose

Manipulates the protection domain (page access as controlled by storage keys) in use for code execution in the kernel environment.

Syntax

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

hkeyset_t hkeyset_add (hkeyset_t *keyset* **);**
hkeyset_t hkeyset_replace (hkeyset_t *keyset* **);**
void hkeyset_restore (hkeyset_t *keyset* **);**
hkeyset_t hkeyset_get (*void* **);**

Parameters

Item	Description
keyset	The hardware keyset to be activated.

Description

If storage protection keys are enabled, every memory page has a hardware storage protection key associated with it. A keyset is a representation of the access rights to a set of storage protection keys. To access a memory page, a hardware keyset containing the storage key associated with the memory page must be active.

The **hkeyset_add** kernel service updates the protection domain by adding the hardware keyset specified by the *keyset* parameter to the currently addressable hardware keyset. The previous hardware keyset is returned.

The **hkeyset_replace** kernel service updates the protection domain by loading the hardware keyset specified by the *keyset* parameter as the currently addressable storage set. The previous hardware keyset is returned.

The **hkeyset_restore** kernel service restores a caller's hardware keyset when returning from a module entry point. It does not return any value.

The **hkeyset_get** kernel service reads the current hardware keyset without altering it.

Execution Environment

The **hkeyset_add**, **hkeyset_replace**, **hkeyset_restore**, or **hkeyset_get** kernel service can be called from either the process environment or the interrupt environment.

Return Values

The **hkeyset_add**, **hkeyset_replace**, and **hkeyset_get** kernel services return the *keyset* value that was active before the call. The **hkeyset_restore** kernel service does not return any value.

Related information:

Kernel Storage Protection Keys Concepts

hkeyset_restore_userkeys Kernel Service

Purpose

Restores the previous user-memory access.

Syntax

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

kerrno_t hkeyset_restore_userkeys (oldset)
hkeyset_t oldset;
```

Parameters

Item	Description
oldset	Specifies the previous hardware keyset returned by the **hkeyset_update_userkeys** kernel service.

Description

The **hkeyset_restore_userkeys** kernel service is a specialized protection gate that restores only the user-mode portion of the current hardware keyset. This is normally done by the kernel after this kernel service accesses user memory.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_HKEYSET_RESTORE_USERKEYS	Indicates that the execution environment is not valid.

Related reference:
"hkeyset_update_userkeys Kernel Service"

hkeyset_update_userkeys Kernel Service

Purpose

Establishes accessibility to user memory.

Syntax

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

kerrno_t hkeyset_update_userkeys (oldset)
hkeyset_t *oldset;
```

Parameters

Item	Description
oldset	Contains the returned previous hardware keyset. The valid parameter must be an 8-byte aligned address.

Description

The **hkeyset_update_userkeys** kernel service is a specialized protection gate that alters only the user-mode portion of the current hardware keyset. The user-mode storage keys for the currently running thread is placed into the current hardware keyset. This is normally done by the kernel when this kernel service accesses user memory.

The previous hardware keyset is returned in the memory specified by the *oldset* parameter. You can use the **hkeyset_restore_userkeys** kernel service to remove the user accessibility when it is no longer needed.

Important: Kernel services such as **xmemin**, **xmemout**, **uiomove**, **copyin**, and **coypout** are the suggested ways to access user memory from the kernel. If possible, avoid using kernel code to directly access user memory.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_HKEYSET_UPDATE_USERKEYS	Indicates that the parameter or execution environment is not valid.

Related reference:

"hkeyset_restore_userkeys Kernel Service" on page 204

"xmemin Kernel Service" on page 607

"uiomove Kernel Service" on page 516

i

The following kernel services begin with the with the letter i.

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

Item	Description
handler	Specifies the address of the interrupt handler structure passed to the **i_init** service.

Description

The **i_clear** service removes the interrupt handler specified by the *handler* parameter from the set of interrupt handlers that the kernel knows about. "Coding an Interrupt Handler" in *Kernel Extensions and Device Support Programming Concepts* contains a brief description of interrupt handlers.

The **i_mask** service is called by the **i_clear** service to disable the interrupt handler's bus interrupt level when this is the last interrupt handler for the bus interrupt level. The **i_clear** service removes the interrupt handler structure from the list of interrupt handlers. The kernel maintains this list for that bus interrupt level.

Execution Environment

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

Return Values

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

Related reference:

"i_init Kernel Service" on page 216

Related information:

I/O Kernel Services

Understanding Interrupts

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

Item	Description
new	Specifies the new interrupt priority.

Description

> **Attention:** The **i_disable** service has two side effects that result from the replaceable and pageable nature of the kernel. First, it prevents process dispatching. Second, it ensures, within limits, that the caller's stack is in memory. Page faults that occur while the interrupt priority is not equal to **INTBASE** crash the system.

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

The **i_disable** service sets the interrupt priority to a more favored interrupt priority. The interrupt priority is used to control which interrupts are allowed.

A value of **INTMAX** is the most favored priority and disables all interrupts. A value of **INTBASE** is the least favored and disables only interrupts not in use. The **/usr/include/sys/intr.h** file defines valid interrupt priorities.

The interrupt priority is changed only to serialize code executing in more than one environment (that is, process and interrupt environments).

For example, a device driver typically links requests in a list while executing under the calling process. The device driver's interrupt handler typically uses this list to initiate the next request. Therefore, the device driver must serialize updating this list with device interrupts. The **i_disable** and **i_enable** services provide this ability. The **I_init** kernel service contains a brief description of interrupt handlers.

Note: When serializing such code in a multiprocessor-safe kernel extension, locking must be used as well as interrupt control. For this reason, new code should call the **disable_lock** kernel service instead of **i_disable**. The **disable_lock** service performs locking only on multiprocessor systems, and helps ensure that code is portable between uniprocessor and multiprocessor systems.

The **i_disable** service must always be used with the **i_enable** service. A routine must always return with the interrupt priority restored to the value that it had upon entry.

The **i_mask** service can be used when a routine must disable its device across a return.

Because of these side effects, the caller of the **i_disable** service should ensure that:

- The reference parameters are pinned.
- The code executed during the disable operation is pinned.
- The amount of stack used during the disable operation is less than 1KB.
- The called programs use less than 1KB of stack.

In general, the caller of the **i_disable** service should also call only services that can be called by interrupt handlers. However, processes that call the **i_disable** service can call the **e_sleep**, **e_wait**, **e_sleepl**, **lockl**, and **unlockl** services as long as the event word or lockword is pinned.

The kernel's first-level interrupt handler sets the interrupt priority for an interrupt handler before calling the interrupt handler. The interrupt priority for a process is set to **INTBASE** when the process is created and is part of each process's state. The dispatcher sets the interrupt priority to the value associated with the process to be executed.

Execution Environment

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

Return Value

The **i_disable** service returns the current interrupt priority that is subsequently used with the **i_enable** service.

Related reference:

"disable_lock Kernel Service" on page 76

"i_enable Kernel Service"

Related information:

Understanding Interrupts

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

Item	Description
old	Specifies the interrupt priority returned by the **i_disable** service.

Description

The **i_enable** service restores the interrupt priority to a less-favored value. This value should be the value that was in effect before the corresponding call to the **i_disable** service.

Note: When serializing a thread with an interrupt handler in a multiprocessor-safe kernel extension, locking must be used as well as interrupt control. For this reason, new code should call the **unlock_enable** kernel service instead of **i_enable**. The **unlock_enable** service performs locking only on multiprocessor systems, and helps ensure that code is portable between uniprocessor and multiprocessor systems.

Execution Environment

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

Return Values

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

Related reference:

"i_disable Kernel Service" on page 207

"unlock_enable Kernel Service" on page 517

Related information:

Understanding Interrupts

i_eoi Kernel Service

Purpose

Issues an End of Interrupt (EOI) for a given handler.

Syntax

```
int i_eoi(struct intr   *handler)
```

Description

The **i_eoi** kernel service allows a device driver to issue an End of Interrupt (EOI) for its device explicitly. For level-triggered interrupts, after the second level interrupt handler (SLIH) has completed, the kernel issues an EOI on behalf of the device driver. For ISA (8259) edge-triggered interrupts, the kernel issues the EOI on behalf of the device driver before calling the SLIH. However, in the case of some edge-triggered interrupts (for example, PCI and PCI-E style edge-triggered interrupt), it is desirable that the device driver checks for pending work before the EOI is issued, and the driver is required to check for any additional work after the EOI is issued. The **i_eoi** kernel service facilitates such operations and issues an EOI for an edge-triggered interrupt source. The **i_eoi** kernel service fails if called for a level-triggered interrupt source.

Parameters

Item	Description
handler	Pointer to the interrupt handler

Execution Environment

The **i_eoi** kernel service can be called from process or interrupt environment.

Return Values

INTR_SUCC if successful

INTR_FAIL if unsuccessful (the **INTR_EDGE** flag was not set on **i_init**()).

Virtual device drivers' interrupt services are similar to the PCI interrupt services. Interrupts are registered with a **bus_type** of **BUS_BID**. The primary difference is that the edge flag should be set for vdevices. For example:

```
Parent  CuDv  "bus_id"  VDEVICE bus BID
Device  CuAt  "bus_intr_lvl" Adapter interrupt level

intr.flags |= INTR_EDGE
intr.bus_type = BUS_BID
intr.bid = Parent_CuDv.bus_id
intr.level = Device_CuAt.bus_intr_lvl
```

PCI-E interrupts are Message Signalled Interrupts, and hence, they are edge-triggered. Therefore, **INTR_EDGE** flag should be specified.

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

Item	Description
addr	Specifies a complete address.

Description

The **ifa_ifwithaddr** kernel service is passed a complete address and locates the corresponding interface. If successful, the **ifa_ifwithaddr** service returns the **ifaddr** structure associated with that address.

Execution Environment

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

Return Values

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

Example

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

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

Related reference:

"ifa_ifwithdstaddr Kernel Service"

"ifa_ifwithnet Kernel Service" on page 212

Related information:

Network Kernel Services

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

Item	Description
addr	Specifies a destination address.

Description

The **ifa_ifwithdstaddr** kernel service searches the list of point-to-point addresses per interface and locates the connection with the destination address specified by the *addr* parameter.

Execution Environment

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

Return Values

If successful, the **ifa_ifwithdstaddr** service returns the corresponding **ifaddr** structure associated with the point-to-point interface. If no interface is found, the **ifa_ifwithdstaddr** service returns a null pointer.

Example

To locate the point-to-point interface with a given destination address, invoke the **ifa_ifwithdstaddr** kernel service as follows:

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

Related reference:

"ifa_ifwithaddr Kernel Service" on page 210

"ifa_ifwithnet Kernel Service"

Related information:

Network Kernel Services

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

Item	Description
addr	Specifies the address.

Description

The **ifa_ifwithnet** kernel service locates an interface that matches the network specified by the address it is passed. If more than one interface matches, the **ifa_ifwithnet** service returns the first interface found.

Execution Environment

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

Return Values

If successful, the **ifa_ifwithnet** service returns the **ifaddr** structure of the correct interface. If no interface is found, the **ifa_ifwithnet** service returns a null pointer.

Example

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

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

Related reference:

"ifa_ifwithaddr Kernel Service" on page 210

"ifa_ifwithdstaddr Kernel Service" on page 211

Related information:

Network Kernel Services

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

Item	Description
ifp	Points to the interface network (**ifnet**) structure that defines the network interface.

Description

The **if_attach** kernel service registers a Network Interface Driver (NID) in the network interface list.

Execution Environment

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

Return Values

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

Related reference:

"if_detach Kernel Service"

Related information:

Network Kernel Services

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

Item	Description
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

Item	Description
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.

Related reference:

"if_attach Kernel Service" on page 212

Related information:

Network Kernel Services

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

Item	Description
ifp	Specifies the **ifnet** structure associated with the interface array.

Description

The **if_down** kernel service:

- Marks an interface as down by setting the `flags` field of the **ifnet** structure appropriately.
- Notifies the protocols of the transaction.
- Flushes the output queue.

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

Execution Environment

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

Return Values

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

Example

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

```
if_down(ifp);
```

Related information:

Network Kernel Services

if_nostat Kernel Service

Purpose

Zeroes statistical elements of the interface array in preparation for an attach operation.

Syntax

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

void if_nostat ( ifp)
struct ifnet *ifp;
```

Parameter

Item	Description
ifp	Specifies the **ifnet** structure associated with the interface array.

Description

The **if_nostat** kernel service zeroes the statistic elements of the **ifnet** structure for the interface. The *ifp* parameter specifies the **ifnet** structure associated with the interface that is being attached. The **if_nostat** service is called from the interface attach routine.

Execution Environment

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

Return Values

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

Example

To zero statistical elements of the interface array in preparation for an attach operation, invoke the **if_nostat** kernel service as follows:

```
if_nostat(ifp);
```

Related information:

Network Kernel Services

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

Item	Description
name	Specifies the name of an interface (for example, en0).

Description

The **ifunit** kernel service searches the list of configured interfaces for an interface specified by the *name* parameter. If a match is found, the **ifunit** service returns the address of the **ifnet** structure for that interface.

Execution Environment

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

Return Values

The **ifunit** kernel service returns the address of the **ifnet** structure associated with the named interface. If the interface is not found, the service returns a null value.

Example

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

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

Related information:

Network Kernel Services

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

Item	Description
handler	Designates the address of the pinned interrupt handler structure.

Description

> **Attention:** The interrupt handler structure must not be altered between the call to the **i_init** service to define the interrupt handler and the call to the **i_clear** service to remove the interrupt handler. The structure must also stay pinned. If this structure is altered at those times, a kernel panic may result.

The **i_init** service allows device drivers to define an interrupt handler to the kernel. The interrupt handler **intr** structure pointed to by the *handler* parameter describes the interrupt handler. The caller of the **i_init** service must initialize all the fields in the **intr** structure. The **/usr/include/sys/intr.h** file defines these fields and their valid values.

The **i_init** service enables interrupts by linking the interrupt handler structure to the end of the list of interrupt handlers defined for that bus level. If this is the first interrupt handler for the specified bus interrupt level, the **i_init** service enables the bus interrupt level by calling the **i_unmask** service.

The interrupt handler can be called before the **i_init** service returns if the following two conditions are met:

- The caller of the **i_init** service is executing at a lower interrupt priority than the one defined for the interrupt.
- An interrupt for the device or another device on the same bus interrupt level is already pending.

On multiprocessor systems, all interrupt handlers defined with the **i_init** kernel service run by default on the first processor started when the system was booted. This ensures compatibility with uniprocessor interrupt handlers. If the interrupt handler being defined has been designed to be multiprocessor-safe, or is an EPOW (Early Power-Off Warning) or off-level interrupt handler, set the **INTR_MPSAFE** flag in the `flags` field of the **intr** structure passed to the **i_init** kernel service. The interrupt handler will then run on any available processor.

Coding an Interrupt Handler

The kernel calls the interrupt handler when an enabled interrupt occurs on that bus interrupt level. The interrupt handler is responsible for determining if the interrupt is from its own device and processing the interrupt. The interface to the interrupt handler is as follows:

int *interrupt_handler* **(***handler***) struct intr** **handler***;**

The *handler* parameter points to the same interrupt handler structure specified in the call to the **i_init** kernel service. The device driver can pass additional parameters to its interrupt handler by declaring the interrupt handler structure to be part of a larger structure that contains these parameters.

The interrupt handler can return one of two return values. A value of **INTR_SUCC** indicates that the interrupt handler processed the interrupt and reset the interrupting device. A value of **INTR_FAIL** indicates that the interrupt was not from this interrupt handler's device.

Registering Early Power-Off Warning (EPOW) Routines

The **i_init** kernel service can also be used to register an EPOW (Early Power-Off Warning) notification routine.

The return value from the EPOW interrupt handler should be **INTR_SUCC**, which indicates that the interrupt was successfully handled. All registered EPOW interrupt handlers are called when an EPOW interrupt is indicated.

Execution Environment

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

Return Values

Item	Description
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.

Related information:

Understanding Interrupts

I/O Kernel Services

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

Item	Description
handler	Specifies the address of the interrupt handler structure that was passed to the **i_init** service.

Description

The **i_mask** service disables the bus interrupt level specified by the *handler* parameter.

The **i_disable** and **i_enable** services are used to serialize the execution of various device driver routines with their device interrupts.

The **i_init** and **i_clear** services use the **i_mask** and **i_unmask** services internally to configure bus interrupt levels.

Device drivers can use the **i_disable**, **i_enable**, **i_mask**, and **i_unmask** services when they must perform off-level processing with their device interrupts disabled. Device drivers also use these services to allow process execution when their device interrupts are disabled.

Execution Environment

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

Return Values

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

Related reference:

"i_unmask Kernel Service" on page 238

Related information:

Understanding Interrupts

in_localaddr Kernel Service

Purpose

Determine whether an IPv4 address is on the local network

Syntax

```
#include <arpa/inet.h>

int in_localaddr (struct in_addr in)
```

Parameters

in Specifies the IPv4 address

Description

Indicates that the IPv4 address in is for a local host (one to which we have a connection). If subnetsarelocal is true, this includes other subnets of the local net. Otherwise, it includes only the directly-connected (sub)nets.

Execution Environment

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

Return Values

1 Indicates that the internet address is for a local host (one to which we have a connection). If `subnetsarelocal` is true, this includes other subnets of the local net. Otherwise, it includes only the directly-connected (sub)nets.

init_heap Kernel Service

Purpose

Initializes a new heap to be used with kernel memory management services.

Syntax

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

heapaddr_t init_heap ( area,  size,  heapp)
caddr_t area;
int size;
heapaddr_t *heapp;
```

Parameters

Item	Description
area	Specifies the virtual memory address used to define the starting memory area for the heap. This address must be page-aligned.
size	Specifies the size of the heap in bytes. This value must be an integral number of system pages.
heapp	Points to the external heap descriptor. This must have a null value. The base kernel uses this field is used to specify special heap characteristics that are unavailable to kernel extensions.

Description

The **init_heap** kernel service is most commonly used by a kernel process to initialize and manage an area of virtual memory as a private heap. Once this service creates a private heap, the returned **heapaddr_t** value can be used with the **xmalloc** or **xmfree** service to allocate or deallocate memory from the private heap. Heaps can be created within other heaps, a kernel process private region, or even on a stack.

Few kernel extensions ever require the **init_heap** service because the exported global **kernel_heap** and **pinned_heap** are normally used for memory allocation within the kernel. However, kernel processes can use the **init_heap** service to create private nonglobal heaps within their process private region for controlling kernel access to the heap and possibly for performance considerations.

Execution Environment

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

Related reference:

"xmalloc Kernel Service" on page 597

Related information:

Memory Kernel Services

Using Kernel Processes

initp Kernel Service

Purpose

Changes the state of a kernel process from idle to ready.

Syntax

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

int initp
(pid, func, init_parms,
parms_length, name)
pid_t  pid;
void ( func) (int
flag, void* init_parms, int parms_length );
void * init_parms;
int  parms_length;
char * name;
```

Parameters

Item	Description
pid	Specifies the process identifier of the process to be initialized.
func	Specifies the process's initialization routine.
init_parm	Specifies the pointer to the initialization parameters.
parms_length	Specifies the length of the initialization parameters.
name	Specifies the process name.

Description

The **initp** kernel service completes the transition of a kernel process from idle to ready. The idle state for a process is represented by **p_status == SIDL**. Before calling the **initp** service, the **creatp** service is called to create the process. The **creatp** service allocates and initializes a process table entry.

The **initp** service creates and initializes the process-private segment. The process is marked as a kernel process by a bit set in the **p_flag** field in the process table entry. This bit, the SKPROC bit, signifies that the process is a kernel process.

The process calling the **initp** service to initialize a newly created process must be the same process that called the **creatp** service to create the new process.

"Using Kernel Processes" in *Kernel Extensions and Device Support Programming Concepts* further explains how the **initp** kernel service completes the initialization process begun by the **creatp** service.

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

The *name* parameter points to a character string that names the process. The leading characters of this string are copied to the user structure. The number of characters copied is implementation-dependent, but at least four are always copied.

The *func* parameter indicates the main entry point of the process. The new process is made ready to run this function. If the *init_parms* parameter is not null, it points to data passed to this routine. The parameter structure must be agreed upon between the initializing and initialized process. The **initp** service copies the data specified by the *init_parm* parameter (with the exact number of bytes specified by the *parms_length* parameter) of data to the new process's stack.

Execution Environment

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

Example

To initialize the kernel process running the function *main_kproc*, enter:

```
{
.
.
.
pid = creatp();
initp(pid, main_kproc, &node_num, sizeof(int), "tkproc");
.
.
}
void
main_kproc(int flag, void* init_parms, int parms_length)
{
        .
        .
        .
        int i;
        i = *( (int *)init_parms );
        .
        .
        .
}
```

Return Values

Item	Description
0	Indicates a successful operation.
ENODEV	The process could not be scheduled because it has a processor attachment that does not contain any available processors. This can be caused by Dynamic Processor Deallocation.
ENOMEM	Indicates that there was insufficient memory to initialize the process.
EINVAL	Indicates an *pid* parameter that was not valid.

Related reference:

"creatp Kernel Service" on page 53

Related information:

Process and Exception Management Kernel Services

Dynamic logical partitioning

initp Kernel Service func Subroutine

Purpose

Directs the process initialization routine.

Syntax

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

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

Parameters

Item	Description
func	Specifies the process's initialization routine.
flag	Has a 0 value if this subroutine is executed as a result of initializing a process with the **initp** service.
init_parms	Specifies the pointer to the initialization parameters.
parms_length	Specifies the length of the initialization parameters.

Related reference:

"initp Kernel Service" on page 220

Related information:

Process and Exception Management Kernel Services

io_map Kernel Service

Purpose

Attach to an I/O mapping

Syntax

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

void *  io_map (io_handle)
io_handle_t io_handle;
```

Description

The **io_map** kernel service sets up addressibility to the I/O address space defined by the **io_handle_t** structure. It returns an effective address representing the start of the mapped region.

The **io_map** kernel service is a replacement call for the **iomem_att** kernel service, which is deprecated on AIX 6.1. However, the **io_map** kernel service might replace multiple **iomem_att** calls depending on the device, the driver, and whether multiple regions were mapped into a single virtual segment. Like the **iomem_att** kernel service, this service does not return any kind of failure. If something goes wrong, the system crashes.

There is a major difference between **io_map** and **iomem_att**. **iomem_att** took an **io_map** structure containing a bus address and returned a fully qualified effective address with any byte offset from the bus address preserved and computed into the returned effective address. The **io_map** kernel service always returns a segment-aligned effective address representing the beginning of the I/O segment corresponding to **io_handle_t**. Manipulation of page and byte offsets within the segment are responsibilities of the device driver.

The **io_map** kernel service is subject to nesting rules regarding the number of attaches allowed. A total system number of active temporary attaches is 4. However, it is recommended that no more than one active attach be owned by a driver calling the interrupt or DMA kernel services. It is also recommended that no active attaches be owned by a driver when calling other kernel services.

Parameters

Item	Description
io_handle	Received on a prior successful call to io_map_init. Describes the I/O space to attach to.

Execution Environment

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

Return Values

The **io_map** kernel service returns a segment-aligned effective address to access the I/O address spaces.

Related reference:

"io_map_init Kernel Service" on page 224

"io_unmap Kernel Service" on page 225

Related information:

Programmed I/O (PIO) Kernel Services

io_map_clear Kernel Service

Purpose

Removes an I/O mapping segment.

Syntax

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

void io_map_clear (io_handle)
io_handle_t io_handle;
```

Description

This service destroys all mappings defined by the *io_handle_t* parameter.

There should be no active mappings (outstanding **io_map** calls) to this handle when **io_map_clear** is called. The segment previously created by an **io_map_init** call or multiple **io_map_init** calls, is deleted.

Parameters

Item	Description
io_handle	Received on a prior successful call to io_map_init. Describes the I/O space to be removed.

Execution Environment

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

Related reference:

"io_map Kernel Service" on page 222

"io_unmap Kernel Service" on page 225

Related information:

Programmed I/O (PIO) Kernel Services

io_map_init Kernel Service

Purpose

Creates and initializes an I/O mapping segment.

Syntax

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

io_handle_t io_map_init (io_map_ptr, page_offset, io_handle)
struct io_map *io_map_ptr;
vpn_t page_offset;
io_handle_t io_handle;

struct io_map {
        int key;                        /* structure version number */
        int flags;                      /* flags for mapping */
        int32long64_t size;            /* size of address space needed */
        int bid;                        /* bus ID */
        long long busaddr;             /* bus address */
};
```

Description

The **io_map_init** kernel service will create a segment to establish a cache-inhibited virtual-to-real translation for the bus address region defined by the contents of the **io_map** struct. The *flags* parameter of the **io_map** structure can be used to customize the mapping such as making the region read-only, using the **IOM_RDONLY** flag.

The **io_map_init** kernel service returns a handle of an opaque type *io_handle_t* to be used on future **io_map** or **io_unmap** calls. All services that use the *io_handle* returned by **io_map_init** must use the handle from the most recent call. Using an old handle is a programming error.

The *vpn_t* type parameter represents the virtual page number offset to allow the caller to specify where, in the virtual segment, to map this region. The offset must not conflict with a previous mapping in the segment. The caller should map the most frequently accessed and performance critical I/O region at *vpn_t* offset 0 into the segment. This is due to the fact that the subsequent **io_map** calls using this *io_handle* will return an effective address representing the start of the segment (that is, page offset 0). The device driver is responsible for managing various offsets into the segment. A single bus memory address page can be mapped multiple times at different *vpn_t* offsets within the segment.

The *io_handle_t* parameter is useful when the caller wants to append a new mapping to an existing segment. For the initial creation of a new I/O segment, this parameter must be NULL. For appended mappings to the same segment, this parameter is the *io_handle_t* returned from the last successful **io_map_init** call. If the mapping fails for any reason (offset conflicts with prior mapping, or no more room in the segment), NULL is returned. In this case, the previous *io_handle_t* is still valid. If successful, the *io_handle_t* returned should be used on all future calls. In this way, a device driver can manage multiple I/O address spaces of a single adapter within a single virtual address segment, requiring the driver to do only a single attach, io_map, to gain addressibility to all of the mappings.

Parameters

Item	Description
io_map_ptr	Pointer to **io_map** structure describing the address region to map.
page_offset	Page offset at which to map the specified region into the virtual address segment.
io_handle	For the first call, this parameter should be NULL. When adding to an existing mapping, this parameter is the *io_handle* received on a prior successful call to **io_map_init**.

Execution Environment

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

Return Values

Item	Description
io_handle_t	An opaque handle to the mapped I/O segment in the virtual memory that must be used in subsequent calls to this service.
NULL	Failed to create or append mapping.

Related reference:

"io_map_clear Kernel Service" on page 223

"io_unmap Kernel Service"

Related information:

Programmed I/O (PIO) Kernel Services

io_unmap Kernel Service

Purpose

Detach from an I/O mapping

Syntax

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

void io_unmap (eaddr)
void *eaddr;
```

Description

The **io_unmap** kernel service removes addressibility to the I/O address space defined by the *eaddr* parameter. There must be a valid active mapping from a previous **io_map** call for this effective address. The *eaddr* parameter can be any valid effective address within the segment, and it does not have to be exactly the same as the address returned by **io_map**.

The **io_unmap** kernel service is a replacement call for the **iomem_det** kernel service, which is deprecated on AIX 6.1. However, the **io_unmap** kernel service might replace multiple **iomem_det** calls depending on the device, the driver, and whether multiple regions were mapped into a single virtual segment using the **io_map_init** kernel service.

Parameters

Item	Description
eaddr	Received on a prior successful call to io_map. Effective address for the I/O space to detach from.

Execution Environment

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

Related reference:

"io_map_clear Kernel Service" on page 223

"io_map Kernel Service" on page 222

Related information:

Programmed I/O (PIO) Kernel Services

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

Item	Description
bp	Specifies the address of the **buf** structure for the buffer whose I/O has completed. On a platform that supports storage keys, the passed in *bp* parameter must be in the **KKEY_PUBLIC** or **KKEY_BLOCK_DEV** protection domain.

Description

A device driver calls the **iodone** kernel service when a block I/O request is complete. The device driver must not reference or alter the buffer header or buffer after calling the **iodone** service.

The **iodone** service takes one of two actions, depending on the current interrupt level. Either it invokes the caller's individual **iodone** routine directly, or it schedules I/O completion processing for the buffer to be performed off-level, at the **INTIODONE** interrupt level. The interrupt handler for this level then calls the iodone routine for the individual device driver. In either case, the individual iodone routine is defined by the `b_iodone` buffer header field in the buffer header. This iodone routine is set up by the caller of the device's strategy routine.

For example, the file I/O system calls set up a routine that performs buffered I/O completion processing. The **uphysio** service sets up a routine that performs raw I/O completion processing. Similarly, the pager sets up a routine that performs page-fault completion processing.

Setting up an iodone Routine

Under certain circumstances, a device driver can set up an **iodone** routine. For example, the logical volume device driver can follow this procedure:

1. Take a request for a logical volume.
2. Allocate a buffer header.
3. Convert the logical volume request into a physical volume request.
4. Update the allocated buffer header with the information about the physical volume request. This includes setting the `b_iodone` buffer header field to the address of the individual iodone routine.
5. Call the physical volume device driver strategy routine.

   Here, the caller of the logical volume strategy routine has set up an iodone routine that is started when the logical volume request is complete. The logical volume strategy routine in turn sets up an iodone routine that is invoked when the physical volume request is complete.

The key point of this example is that only the caller of a strategy routine can set up an iodone routine and even then, this can only be done while setting up the request in the buffer header.

The interface for the **iodone** routine is identical to the interface to the **iodone** service.

Execution Environment

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

Return Values

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

Related reference:

"iowait Kernel Service" on page 232

"buf Structure" on page 614

Related information:

I/O Kernel Services

iostadd Kernel Service

Purpose

Registers an I/O statistics structure that is 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;
```

Description

The **iostadd** kernel service is used to register the I/O statistics structure that is required to maintain statistics on a device. The **iostadd** service is typically called by a tty, disk, or CD-ROM device driver to provide the statistical information that is used by the **iostat** subroutine. The **iostat** subroutine displays statistic information for tty and disk devices on the system. The **iostadd** service must be used once for each configured device.

The **iostadd** kernel service and the **dkstat** structure support Multi-Path I/O (MPIO). For an MPIO device, the anchor is the disk's **dkstat** structure. This anchor must be the first **dkstat** structure that is registered by using the **iostadd** kernel service. Any path **dkstat** structures that are registered later must reference the address of the anchor **dkstat** (disk) structure in the `dkstat.dk_mpio_anchor` field.

For tty devices, the *devtype* parameter has a value of **DD_tty**. In this case, the **iostadd** service uses the *devstatp* parameter to return a pointer to a **ttystat** structure.

For disk or CD-ROM devices with a *devtype* value of **DD_DISK** or **DD_CD-ROM**, the caller must provide a pinned and initialized **dkstat** structure as an input parameter. This structure is pointed to by the *devstatp* parameter on entry to the **iostadd** kernel service.

If the device driver support for a device is terminated, the **dkstat** or **ttystat** structure that is registered with the **iostadd** kernel service must be deregistered by calling the **iostdel** kernel service.

I/O Statistics Structures

The **iostadd** kernel service uses two structures that are found in the **usr/include/sys/iostat.h** file: the **ttystat** structure and the **dkstat** structure.

The **ttystat** structure contains the following fields:

Field	Description
rawinch	Count of raw characters that are received by the tty device
caninch	Count of canonical characters that are generated from canonical processing
outch	Count of the characters output to a tty device

The second structure that is used by the **iostadd** kernel service is the **dkstat** structure, which contains information about disk devices. This structure contains the following fields:

Field	Description
diskname	32-character string name for the disk's logical device
dknextp	Pointer to the next **dkstat** structure in the chain
dk_status	Disk entry-status flags
dk_time	Time the disk is active
dk_bsize	Number of bytes in a block
dk_xfers	Number of transfers to or from the disk
dk_rblks	Number of blocks that are read from the disk
dk_wblks	Number of blocks that are written to the disk
dk_seeks	Number of seek operations for disks
dk_version	Version of the **dkstat** structure
dk_q_depth	Queue depth
dk_mpio_anchor	Pointer to the path data anchors (disk)
dk_mpio_next_path	Pointer to the next path **dkstat** structure in the chain
dk_mpio_path_id	Path ID

tty Device Driver Support

The `rawinch` field in the **ttystat** structure must be incremented by the number of characters that are received by the tty device. The `caninch` field in the **ttystat** structure must be incremented by the number of input characters that are generated from canonical processing. The `outch` field is increased by the number of characters output to tty devices. These fields must be incremented by the device driver, but never be cleared.

Disk Device Driver Support

A disk device driver must perform these four tasks:

- Allocate and pin a **dkstat** structure during device initialization.
- Update the `dkstat.diskname` field with the device's logical name.
- Update the `dkstat.dk_bsize` field with the number of bytes in a block on the device.
- Set all other fields in the structure to 0.

If a disk device driver supports MPIO, it must perform the following 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 the value of `dkstat.dk_version` to **dk_qd_mpio_magic**.
- Set the value of `dkstat.dk_mpio_anchor` to 0 if the **dkstat** added structure is the disk.
- Set the value of `dkstat.dk_mpio_anchor` to the address of the path's anchor (disk) **dkstat** structure, and set `dkstat.dk_mpio_path_id` to the path's ID if the **dkstat** added structure is a path.
- Set all other fields to 0.

If the device supports discrete seek commands, the `dkstat.dk_xrate` field in the structure must be set to the transfer rate capability of the device (KB/sec). The device's **dkstat** structure must then be registered by using the **iostadd** kernel service.

During drive operation update, the dkstat.dk_status field must show the busy or non-busy state of the device. It can be done by setting and resetting the **IOST_DK_BUSY** flag. The dkstat.dk_xfers field must be incremented for each transfer initiated to or from the device. The dkstat.dk_rblks and dkstat.dk_wblks fields must be incremented by the number of blocks that are read or written.

If the device supports discrete seek commands, the dkstat.dk_seek field must be incremented by the number of seek commands that are sent to the device. If the device does not support discrete seek commands, both the dkstat.dk_seek and dkstat.dk_xrate fields must be left with a value of 0.

The base kernel updates the dkstat.dk_nextp and dkstat.dk_time fields. They mut not be modified by the device driver after initialization. For MPIO devices, the base kernel also updates the dkstat.dk_mpio_next_path field.

Note: The same **dkstat** structure must not be registered more than once.

In addition to basic tasks, a disk driver must perform the following tasks before it calls the **iostadd** kernel service if the driver supports the **-D** option of the **iostat** command:

- Set the value of dkstat.dk_version to **dk_qd_service2_magic**.
- Set the dkstat.ident.adapter field to the adapter name if the driver does not support MPIO.

During I/O operations, the driver must perform the following tasks:

- Increase the dkstat.__dk_rxfers field for each read transfer.
- Update the dkstat.dk_q_depth field with the number of I/O requests which are in progress.
- Increase the dkstat.dk_q_full field when the number of I/O requests which are in progress reaches the maximum queue depth.
- Increase the dkstat.dk_rserv field by the service time which is the delta-time base value between when the **devstrat** kernel service sends a read request to the adapter driver and when the **iodone** kernel service returns the request from the adapter driver.
- Increase the dkstat.dk_rtimeout field when the driver tries a failed read request again.
- Increase the dkstat.dk_rfailed field when the driver returns a failed read request as an error.
- Update the dkstat.dk_min_rserv field with the minimum service time for a read request.
- Update the dkstat.dk_max_rserv field with the maximum service time for a read request.
- Increase the dkstat.dk_wserv field by the service time which is the delta-time base value between when the **devstrat** kernel service sends a write request to the adapter driver and when the **iodone** kernel service returns the request from the adapter driver.
- Increase the dkstat.dk_wtimeout field when the driver tries a failed write request again.
- Increase the dkstat.dk_wfailed field when the driver returns a failed write request as an error.
- Update the dkstat.dk_min_wserv field with the minimum service time for a write request.
- Update the dkstat.dk_max_wserv field with the maximum service time for a write request.
- Increase and decrease the dkstat.dk_wq_depth field when the driver enqueues and dequeues an I/O request.
- Increase the dkstat.dk_wq_time field by the wait time which is the delta-time base value between when the driver enqueues an I/O request and when the **devstrat** kernel service sends the request to the adapter driver.
- Update the dkstat.dk_wq_min_time field with the minimum wait time.
- Update the dkstat.dk_wq_max_time field with the maximum wait time.

Parameters

Item	Description
devtype	Specifies the type of device for which I/O statistics are kept. The various device types are defined in the **/usr/include/sys/devinfo.h** file. Currently, I/O statistics are only kept for disks, CD-ROMs, and tty devices. Possible values for this parameter are: **DD_DISK** For disks **DD_CD-ROM** For CD-ROMs **DD_TTY** For tty devices
devstatp	Points to an I/O statistics structure for the device type that is specified by the *devtype* parameter. For a *devtype* parameter of **DD_tty**, the address of a pinned **ttystat** structure is returned. For a *devtype* parameter of **DD_DISK** or **DD_CD-ROM**, the parameter is an input parameter that points to a **dkstat** structure previously allocated by the caller. On a platform that supports storage keys, the passed in *devstatp* parameter must be in the **KKEY_PUBLIC** or **KKEY_BLOCK_DEV** protection domain.

Execution Environment

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

Return Values

Item	Description
0	Indicates that no error is detected.
EINVAL	Indicates that the *devtype* parameter specified a device type that is not valid. For MPIO devices, indicates that an anchor for a path **dkstat** structure was not found.

Related reference:

"iostdel Kernel Service"

Related information:

iostat subroutine

Kernel Extension and Device Driver Management Kernel Services

iostdel Kernel Service

Purpose

Removes the registration of an I/O statistics structure that is 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;
```

Description

The **iostdel** kernel service removes the registration of an I/O statistics structure for a terminating device. The device's **ttystat** or **dkstat** structure must be previously registered by using the **iostadd** kernel service. Following a return from the **iostdel** service, the **iostat** command no longer displays statistics for the device that is terminated.

The **iostdel** kernel service supports Multi-Path I/O (MPIO). For an MPIO device, the anchor is the disk's **dkstat** structure. An anchor (disk) might have several paths that are associated with it. Each of these paths can have a **dkstat** structure that is registered by using the **iostadd** kernel service. The semantics for unregistering a **dkstat** structure for an MPIO device are more restrictive than for a non-MPIO device. All paths must unregister before the anchor (disk) is unregistered. If the anchor (disk) **dkstat** structure is unregistered before all of the paths that are associated with it are unregistered, the **iostdel** kernel service removes the registration of the anchor (disk) **dkstat** structure and all remaining registered paths.

Parameters

Item	Description
devstatp	Points to an I/O statistics structure previously registered by using the **iostadd** kernel service. On a platform that supports storage keys, the passed in *devstatp* parameter must be in the **KKEY_PUBLIC** or **KKEY_BLOCK_DEV** protection domain.

Execution Environment

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

Return Values

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

Related reference:

“iostadd Kernel Service” on page 228

Related information:

iostat command

Kernel Extension and Device Driver Management Kernel Services

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

Item	Description
bp	Specifies the address of the **buf** structure for the buffer with in-process I/O. On a platform that supports storage keys, the passed in *bp* parameter must be in the **KKEY_PUBLIC** or **KKEY_BLOCK_DEV** protection domain.

Description

The **iowait** kernel service causes a process to wait until the I/O is complete for the buffer specified by the *bp* parameter. Only the caller of the strategy routine can call the **iowait** service. The **B_ASYNC** bit in the buffer's `b_flags` field should not be set.

The **iodone** kernel service must be called when the block I/O transfer is complete. The **buf** structure pointed to by the *bp* parameter must specify an iodone routine. This routine is called by the iodone interrupt handler in response to the call to the **iodone** kernel service. This iodone routine must call the **e_wakeup** service with the `bp->b_events` field as the event. This action awakens all processes waiting on I/O completion for the **buf** structure using the **iowait** service.

Execution Environment

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

Return Values

The **iowait** service uses the **geterror** service to determine which of the following values to return:

Item	Description
0	Indicates that I/O was successful on this buffer.
EIO	Indicates that an I/O error has occurred.
b_error value	Indicates that an I/O error has occurred on the buffer.

Related reference:
"geterror Kernel Service" on page 186
"iodone Kernel Service" on page 226
"buf Structure" on page 614

ip_fltr_in_hook, ip_fltr_out_hook, ipsec_decap_hook, inbound_fw, outbound_fw Kernel Service

Purpose

Contains hooks for IP filtering.

Syntax

```
#define FIREWALL_OK        0  /* Accept IP packet                    */
#define FIREWALL_NOTOK     1  /* Drop IP packet                      */
#define FIREWALL_OK_NOTSEC 2  /* Accept non-encapsulated IP packet
                                 (ipsec_decap_hook only)      */
#include <sys/mbuf.h>
#include <net/if.h>

int (*ip_fltr_in_hook)(struct mbuf **pkt, void **arg)

int (*ipsec_decap_hook)(struct mbuf **pkt, void **arg)

int (*ip_fltr_out_hook)(struct ifnet *ifp, struct mbuf **pkt,
int flags)
```

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

#include <sys/mbuf.h>

#include <netinet/ip_var.h>

void (*inbound_fw)(struct ifnet *ifp, struct mbuf *m,
inbound_fw_args_t *args)

void ipintr_noqueue_post_fw(struct ifnet *ifp, struct mbuf *m,
inbound_fw_args_t *args)

inbound_fw_args_t *inbound_fw_save_args(inbound_fw_args_t *args)

int (*outbound_fw)(struct ifnet *ifp, struct mbuf *
m0, outbound_fw_args_t *args)

int ip_output_post_fw( struct ifnet *ifp, struct mbuf *m0,
outbound_fw_args_t *args)

outbound_fw_args_t *outbound_fw_save_args(outbound_fw_args_t *args)
```

Parameters

Item	Description
pkt	Points to the mbuf chain containing the IP packet to be received (**ip_fltr_in_hook**, **ipsec_decap_hook**) or transmitted **(ip_fltr_out_hook**). The *pkt* parameter may be examined and/or changed in any of the three hook functions.
arg	Is the address of a pointer to *void* that is locally defined in the function where **ip_fltr_in_hook** and **ipsec_decap_hook** are called. The *arg* parameter is initially set to NULL, but the address of this pointer is passed to the two hook functions, **ip_fltr_in_hook** and **ipsec_decap_hook**. The *arg* parameter may be set by either of these functions, thereby allowing a void pointer to be shared between them.
ifp	Is the outgoing interface on which the IP packet will be transmitted for the **ip_fltr_out_hook** function.
flags	Indicates the ip_output flags passed by a transport layer protocol. Valid flags are currently defined in the **/usr/include/netinet/ip_var.h** files. See the Flags section below.

Description

These routines provide kernel-level hooks for IP packet filtering enabling IP packets to be selectively accepted, rejected, or modified during reception, transmission, and decapsulation. These hooks are initially NULL, but are exported by the netinet kernel extension and will be invoked if assigned non-NULL values.

The **ip_fltr_in_hook** routine is used to filter incoming IP packets, the **ip_fltr_out_hook** routine filters outgoing IP packets, and the **ipsec_decap_hook** routine filters incoming encapsulated IP packets.

The **ip_fltr_in_hook** function is invoked for every IP packet received by the host, whether addressed directly to this host or not. It is called after verifying the integrity and consistency of the IP packet. The function is free to examine or change the IP packet (*pkt*) or the pointer shared with **ipsec_decap_hook** (*arg*). The return value of the **ip_fltr_in_hook** indicates whether *pkt* should be accepted or dropped. The return values are described in Expected Return Values below. If *pkt* is accepted (a return value of **FIREWALL_OK**) and it is addressed directly to the host, the **ipsec_decap_hook** function is invoked next. If *pkt* is accepted, but is not directly addressed to the host, it is forwarded if IP forwarding is enabled. If **ip_fltr_in_hook** indicates *pkt* should be dropped (a return value of **FIREWALL_NOTOK**), it is neither delivered nor forwarded.

The **ipsec_decap_hook** function is called after reassembly of any IP fragments (the **ip_fltr_in_hook** function will have examined each of the IP fragments) and is invoked only for IP packets that are directly addressed to the host. The **ipsec_decap_hook** function is free to examine or change the IP packet (*pkt*) or the pointer shared with **ipsec_decap_hook** (*arg*). The hook function should perform decapsulation if necessary, back into *pkt* and return the proper status so that the IP packet can be processed appropriately.

See the Expected Return Values section below. For acceptable encapsulated IP packets (a return value of **FIREWALL_OK**), the decapsulated packet is processed again by jumping to the beginning of the IP input processing loop. Consequently, the decapsulated IP packet will be examined first by **ip_fltr_in_hook** and, if addressed to the host, by **ipsec_decap_hook**. For acceptable non-encapsulated IP packets (a return value of **FIREWALL_OK_NOTSEC**), IP packet delivery simply continues and *pkt* is processed by the transport layer. A return value of **FIREWALL_NOTOK** indicates that *pkt* should be dropped.

The **ip_fltr_out_hook** function is called for every IP packet to be transmitted, provided the outgoing IP packet's destination IP address is NOT an IP multicast address. If it is, it is sent immediately, bypassing the **ip_fltr_out_hook** function. This hook function is invoked after inserting the IP options from the upper protocol layers, constructing the complete IP header, and locating a route to the destination IP address. The **ip_fltr_out_hook** function may modify the outgoing IP packet (*pkt*), but the interface and route have already been assigned and may not be changed. The return value from the **ip_fltr_out_hook** function indicates whether *pkt* should be transmitted or dropped. See the Expected Return Values section below. If *pkt* is not dropped (**FIREWALL_OK**), it's source address is verified to be local and, if *pkt* is to be broadcast, the ability to broadcast is confirmed. Thereafter, *pkt* is enqueued on the interfaces (*ifp*) output queue. If *pk*t is dropped (**FIREWALL_NOTOK**), it is not transmitted and **EACCES** is returned to the process.

The **inbound_fw** and **outbound_fw** firewall hooks allow kernel extensions to get control of packets at the place where IP receives them. If **inbound_fw** is set, **ipintr_noqueue**, the IP input routine, calls **inbound_fw** and then exits. If not, **ipintr_noqueue** calls **ipintr_noqueue_post_fw** and then exits. If the **inbound_fw** hook routine wishes to pass the packet into IP, it can call **ipintr_noqueue_post_fw**. **inbound_fw** may copy its *args* parameter by calling **inbound_fw_save_args**, and may free its copy of its *args* parameter by calling **inbound_fw_free_args**.

Similarly, **ip_output** calls **outbound_fw** if it is set, and calls **ip_output_post_fw** if not. The **outbound_fw** hook can call **ip_output_post_fw** if it wants to send a packet. **outbound_fw** may copy its *args* parameter by calling **outbound_fw_save_args**, and later free its copy of its *args* parameter by calling **outbound_fw_free_args**.

Flags

Item	Description
IP_FORWARDING	Indicates that most of the IP headers exist.
IP_RAWOUTPUT	Indicates that the raw IP header exists.
IP_MULTICAST_OPTS	Indicates that multicast options are present.
IP_ROUTETOIF	Contains bypass routing tables.
IP_ALLOWBROADCAST	Provides capability to send broadcast packets.
IP_BROADCASTOPTS	Contains broadcast options inside.
IP_PMTUOPTS	Provides PMTU discovery options.
IP_GROUP_ROUTING	Contains group routing gidlist.

Expected Return Values

Item	Description
FIREWALL_OK	Indicates that *pkt* is acceptable for any of the filtering functions. It will be delivered, forwarded, or transmitted as appropriate.
FIREWALL_NOTOK	Indicates that *pkt* should be dropped. It will not be received (**ip_fltr_in_hook**, **ipsec_decap_hook**) or transmitted (**ip_fltr_out_hook**).
FIREWALL_OK_NOTSEC	Indicates a return value only valid for the **ipsec_decap_hook** function. This indicates that *pkt* is acceptable according to the filtering rules, but is not encapsulated; *pkt* will be processed by the transport layer rather than processed as a decapsulated IP packet.

Related information:

Network Kernel Services

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

Item	Description
handler	Specifies the address of an interrupt handler structure passed to the **i_init** service.

Description

The **i_reset** service resets the bus interrupt specified by the *handler* parameter. A device interrupt handler calls the **i_reset** service after resetting the interrupt at the device on the bus. See **i_init** kernel service for a brief description of interrupt handlers.

Execution Environment

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

Return Values

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

Related reference:

"i_init Kernel Service" on page 216

Related information:

Understanding Interrupts

I/O Kernel Services

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

Item	Description
handler	Specifies the address of the pinned interrupt handler structure.

Description

The **i_sched** service allows device drivers to schedule some of their work to be processed at a less-favored interrupt priority. This capability allows interrupt handlers to run as quickly as possible, avoiding interrupt-processing delays and overrun conditions. See the **i_init** kernel service for a brief description of interrupt handlers.

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

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

> **Attention:** The caller cannot alter any fields in the **intr** structure from the time the **i_sched** service is called until the kernel calls the off-level routine. The structure must also stay pinned. Otherwise, the system may crash.

The interrupt handler structure pointed to by the *handler* parameter describes an off-level interrupt handler. The caller of the **i_sched** service must set up all fields in the **intr** structure. The **INIT_OFFL***n* macros in the **/usr/include/sys/intr.h** file can be used to initialize the *handler* parameter. The *n* value represents the priority class that the off-level handler should run at. Currently, classes from 0 to 3 are defined.

Use of the **i_sched** service has two additional restrictions:

First, the **i_sched** service will not re-register an **intr** structure that is already registered for off-level handling. Since **i_sched** has no return value, the service will simply return normally without registering the specified structure if it was already registered but not yet executed. The kernel removes the **intr** structure from the registration list immediately prior to calling the off-level handler specified in the structure. It is therefore possible for the off-level handler to use the structure again to register another off-level request.

Care must be taken when scheduling off-level requests from a second-level interrupt handler (SLIH). If the off-level request is already registered but has not yet executed, a second registration will be ignored. If the off-level handler is currently executing, or has already run, a new request will be registered. Users of this service should be aware of these timing considerations and program accordingly.

Second, the kernel uses the `flags` field in the specified **intr** structure to determine if this structure is already registered. This field should be initialized once before the first call to the **i_sched** service and should remain unmodified for future calls to the **i_sched** service.

Note: Off-level interrupt handler path length should not exceed 5,000 instructions. If it does exceed this number, real-time support is adversely affected.

Execution Environment

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

Return Values

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

Related reference:

"i_init Kernel Service" on page 216

Related information:

Understanding Interrupts

I/O Kernel Services

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

Item	Description
handler	Specifies the address of the interrupt handler structure that was passed to the **i_init** service.

Description

The **i_unmask** service enables the bus interrupt level specified by the *handler* parameter.

Execution Environment

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

Return Values

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

Related reference:

"i_init Kernel Service" on page 216

"i_mask Kernel Service" on page 218

Related information:

Understanding Interrupts

IS64U Kernel Service

Purpose

Determines if the current user-address space is 64-bit or not.

Syntax

#include <sys/types.h> #include <sys/user.h> int IS64U

Description

The **IS64U** kernel service returns 1 if the current user-address space is 64-bit. It returns 0 otherwise.

Execution Environment

The **IS64U** kernel service can be called from a process or interrupt handler environment. In either case, it will operate only on the current user-address space.

Return Values

Item	Description
0	The current user-address space is 32-bits.
1	The current user-address space is 64-bits.

Related reference:

"as_att64 Kernel Service" on page 11

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

k

The following kernel services begin with the with the letter k.

k_cpuextintr_ctl Kernel Service

Purpose

Performs CPU external interrupt control related operations.

Syntax

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

kerrno_t k_cpuextintr_ctl (*command* , *cpuset* , *flags***)**
extintctl_t *command***;**
rsethandle_t *cpuset***;**
unit *flags***;**

Description

This kernel services provides means of enabling, disabling, and querying the external interrupt state on the CPUs described by the CPU resource set. Enabling or disabling an CPU external interrupt could affect the external interrupt delivery to the CPU. Normally, on multiple CPU system, external interrupts can be delivered to any running CPU, and the distribution among the CPUs is determined by a predefined method. Any external interrupt can only be delivered to a CPU if its interrupt priority is more favored than the current external interrupt priority of the CPU. When external interrupts are disabled via this interface, any external interrupt priority less favored than **INTMAX** will be blocked until interrupts are enabled again. This kernel service is applicable only on selective hardware types.

Note: Since this kernel service change the way that interrupts are delivered, system performance may be affected. This service guarantees at least one online CPU will have external interrupts enabled for all device interrupts. Any DLPAR CPU removal can fail if the operation breaks this guarantee. On an I/O bound system, one CPU may not be enough to handle all of the external interrupts received by the partition. Performance may suffer when there are not enough CPUs enabled to handle external interrupts.

Parameters

Item **Description**

command

Specifies the operation to the CPU specified by the CPU resource set. One of the following values defined in **<sys/intr.h>** can be used:

The following commands are supported:

- EXTINTDISABLE: Disable external interrupts on the CPUs specified by the CPU resource set.
- EXTINTENABLE: Enable external interrupts on the CPUs specified by the CPU resource set
- QUERYEXTINTDISABLE: Return a CPU resource set containing the CPUs that have external interrupts disabled.
- QUERYEXTINTENABLE: Return a CPU resource set containing the CPUs that have externals interrupt enabled.

cpuset

Reference to a CPU resource set. Upon successful return from this kernel service, the CPUs that have the external interrupt control operation done will be set in the CPU resource set.

The CPUs specified by this *cpuset* parameter are logic CPU ids.

flags

Always set to 0 or **EINVAL_INTR_DIS_BAD_FLAGS** will be returned.

Security

The caller must have root authority with **CAP_NUMA_ATTACH** capability or **PV_KER_CONF** privilege in the RBAC environment.

Execution Environment

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

Return Values

Upon successful completion, the **k_cpuextintr_ctl** kernel service returns a 0. If unsuccessful, one of the following **kerrno** value is returned.

Item	**Description**
kerrno	**Description**
EINVAL_EXTINTR_BAD_COMMAND	The command value is not valid.
EINVAL_EXTINTR_BAD_FLAGS	The flags value is unknown.
EINVAL_EXTINTR_BAD_CPUSET	The *cpuset* references NULL.
EINVAL_EXTINTR _NO_RSET	The *cpuset* is empty.
ENOTSUP_EXTINTR_CALLER	The kernel service is called from the interrupt environment.
ENOSYS_EXTINTR_PLATFORM	This function is not implemented on the platform.
EPERM_EXTINTR_OPER	The caller does not have enough privilege to perform the requested operation.

Note: A return value of success does not necessarily indicate that external interrupts have been enabled or disabled on all of the specified CPUs. For example, if a CPU is not online, then the enable or disable operation will not be performed on that CPU. The caller should check the returned *cpuset* to see which CPUs have this operation successfully done. The **k_cpuextintr_ctl kernel** service will not block DR CPU add/remove operation during the whole period of system call.

kcap_is_set and kcap_is_set_cr Kernel Service

Purpose

Determines if the given capability is present in an effective capability set.

Syntax

```
kcap_is_set (capability)
cap_value_t capability;

kcap_is_set_cr (capability, cred)
cap_value_t capability;
struct ucred *cred;
```

Parameters

Item	Description
capability	Specifies the capability to be examined. Must be one of the capabilities named in the **sys/capabilities.h** header file.
cred	Pointer to the credentials to be examined.

Description

The **kcap_is_set** subroutine determines if the given capability is present in the current process' effective capability set. The **kcap_is_set_cr** subroutine determines if the given capability is present in the effective capability set of the credentials structure referenced by the *cred* parameter. The *cred* parameter must be a valid referenced credentials structure.

Return Values

The **kcap_is_set** and **kcap_is_set_cr** subroutines return 1 if the capability is present. Otherwise, they return 0.

Related information:

Security Kernel Services

kcid_curproc Kernel Service

Purpose

Returns the current workload partition ID associated with the calling process.

Syntax

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

cid_t kcid_curproc ( )

Description

The **kcid_curproc** kernel service returns the workload partition ID associated with the calling process. You can use this service to determine whether the requesting process is operating within a workload partition (WPAR).

Execution Environment

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

Return Values

If the **kcid_curproc** kernel service is successful, it returns the workload partition ID associated with the calling process. If the calling process is not operating within a WPAR, the ID returned is equivalent to the WPAR_GLOBAL definition found in the **wparid.h** header file.

Related reference:

"kwpar_r2vmap_pid Kernel Service" on page 318

"kwpar_v2rmap_pid Kernel Service" on page 326

kcred_genpagvalue Kernel Service

Purpose

Generates a system-wide unique PAG value for a given PAG type.

Syntax

```
int  kcred_genpagvalue(crp, pag_type, pag_value, pag_flags);
cred_t         *crp;
int           pag_type;
uint64_t  *   pag_value;
int           pag_flags;
```

Description

The **kcred_genpagvalue** kernel service generates a new PAG value for a given PAG type. It is essential that for this function to succeed the PAG type must have been previously registered with the operating system using the **kcred_setpagname** kernel service. The scope of the **kcred_genpagvalue** kernel service is limited to maintaining information about the last generated PAG number and accordingly generating a new number. This service optionally stores the PAG value in the **cred** structure. It does not monitor the PAG values stored in the **cred** structure by other means.

The caller must convert a PAG name to a PAG type using the **kcred_getpagid** kernel service prior to invoking the **kcred_genpagvalue** kernel service.

The *pag_flags* parameter with the **PAG_SET_VALUE** value set causes the generated value to be atomically stored in the process's credentials.

The PAG value returned is of size 64 bits. The number of significant bits is determined by the requested PAG type. 32-bit PAGs have 32 significant bits. 64-bit PAGs have 62 significant bits.

Parameters

Item	Description
pag_type	The *pag_type* parameter is the ID value associated with a PAG name.
pag_value	This pointer points to a buffer where the OS will return the newly generated PAG value.
pag_flags	This parameter must be `0` or the value **PAG_SET_VALUE**.

Return Values

A value of `0` is returned upon successful completion. A negative value is returned if unsuccessful.

Error Codes

Item	Description
EINVAL	The PAG value cannot be generated because the named PAG type does not exist as part of the table.
EPERM	The named PAG type is a 32-bit PAG and the caller does not have the SET_PROC_DAC privilege.

Related reference:

"__pag_getid System Call" on page 398

"kcred_getpagid Kernel Service" on page 245

Related information:

genpagvalue Subroutine

kcred_getcap Kernel Service

Purpose

Copies a capability vector from a credentials structure.

Syntax

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

#include <sys/cred.h>

int kcred_getcap ( crp, cap )
struct ucred * cr;
struct __cap_t * cap;
```

Parameters

Item	Description
crp	Pointer to a credentials structure
cap	Capabilities set

Description

The **kcred_getcap** kernel service copies the capability set from the credentials structure referenced by *crp* into *cap*. *crp* must be a valid, referenced credentials structure.

Execution Environment

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

Return Values

Item	Description
0	Success.
-1	An error has occurred.

Related information:

Security Kernel Services

kcred_getgroups Kernel Service

Purpose

Copies the concurrent group set from a credentials structure.

Syntax

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

int kcred_getgroups ( crp, ngroups, groups )
struct ucred * cr;
int ngroups;
gid_t * groups;
```

Parameters

Item	Description
crp	Pointer to a credentials structure
ngroups	Size of the array of group ID values
groups	Array of group ID values

Description

The **kcred_getgroups** kernel service returns up to *ngroups* concurrent group set members from the credentials structure pointed to by *crp*. *crp* must be a valid referenced credentials structure.

Execution Environment

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

Return Values

Item	Description
>= 0	The number of concurrent groups copied to groups.
-1	An error has occurred.

Related information:

Security Kernel Services

kcred_getpag or kcred_getpag64 Kernel Service

Purpose

Copies a process authentication group (PAG) ID from a credentials structure.

Syntax

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

int kcred_getpag ( crp, which, pag )
struct ucred * cr;
int which;
int * pag;

int kcred_getpag64 ( crp, which, pag )
struct ucred * cr;
int which;
uint64 * pag;
```

Parameters

Item	Description
crp	Pointer to a credentials structure
which	PAG ID to get
pag	Process authentication group

Description

The **kcred_getpag** or **kcred_getpag64** kernel service copies the requested PAG from the credentials structure referenced by *crp* into *pag*. The value of *which* must be a defined PAG ID. The PAG ID for the *Distributed Computing Environment* (DCE) is 0. *crp* must be a valid, referenced credentials structure.

Execution Environment

The **kcred_getpag** or **kcred_getpag64** kernel service can be called from the process environment only.

Return Values

Upon successful completion, a value of `0` is returned. Otherwise, a value of `-1` is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The**kcred_getpag** kernel service fails if the following condition is true:

Item	Description
-EOVERFLOW	PAG value is 64-bit (should be using **kcred_getpag64**)

Related information:

Security Kernel Services

kcred_getpagid Kernel Service

Purpose

Returns the PAG identifier for a PAG name.

Syntax

```
int kcred_getpagid (name)
char *name;
```

Description

Given a PAG type name, the **kcred_getpagid** subroutine returns the PAG identifier for that PAG name.

Parameters

Item	Description
name	A pointer to the name of the PAG type whose integer PAG identifer is to be returned.

Return Values

A return value greater than or equal to 0 is the PAG identifier. A value less than 0 indicates an error.

Error Codes

Item	Description
ENOENT	The *name* parameter doesn't refer to an existing PAG entry.

Related reference:

"__pag_getid System Call" on page 398

"__pag_getvalue System Call" on page 399

"kcred_getpagname Kernel Service" on page 246

kcred_getpaginfo Kernel Service

Purpose

Returns a Process Authentication Group (PAG) flags for a given PAG type.

Syntax

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

int kcred_getpaginfo ( type, infop, infosz )
int type;
struct paginfo *  infop
int infosz;
```

Parameters

Item	Description
type	PAG for which the flags are returned
infop	Pointer to PAG info structure
infosz	Size of **paginfo** structure

Description

The **kcred_getpaginfo** kernel service retrieves the flags for the specific PAG type and stores them in a PAG info structure. The value of *type* must be a defined PAG ID. The PAG ID for the Distributed Computing Environment (DCE) is 0. The *infop* parameter must be a valid, referenced PAG info structure of the size specified by *infosz*.

Execution Environment

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

Return Values

A value of `0` is returned upon successful completion. Upon failure, a `-1` is returned and **errno** is set to a value that explains the error.

Related information:

Security Kernel Services

kcred_getpagname Kernel Service

Purpose

Retrieves the name of a PAG.

Syntax

```
int kcred_getpagname (type, buf, size)
int type;
char *buf;
int size;
```

Description

The **kcred_getpagname** kernel service retrieves the name of a PAG type given its integer value.

Parameters

Item	Description
type	The integer valued identifier representing the PAG type.
buf	A **char** * to where the PAG name is copied.
size	An **int** that specifies the size of *buf* in bytes. The size of the buffer must be PAG_NAME_LENGTH_MAX+1.

Return Values

If successful, a 0 is returned. If unsuccessful, an error code value less than 0 is returned. The PAG name associated with *type* is copied into the caller-supplied buffer *buf*.

Error Codes

Item	Description
EINVAL	The value of *id* is less than 0 or greater than the maximum PAG identifier.
ENOENT	There is no PAG associated with *id*.
ENOSPC	The *size* parameter is insufficient to hold the PAG name.

Related reference:

"__pag_getid System Call" on page 398

"__pag_getname System Call" on page 399

"kcred_setpagname Kernel Service" on page 251

kcred_getppriv Kernel Service

Purpose

Copies a privilege vector from a credentials structure.

Syntax

```
#include <sys/priv.h>
#include <sys/cred.h>

int kcred_getppriv (crp, which, privset)
struct ucred *crp;
int which;
privg_t privset;
```

Parameters

Item	Description
crp	Points to a credentials structure.
which	Specifies the privilege set to get.
privset	Specifies the privilege set.

Description

The **kcred_getppriv** kernel service returns a single privilege set from the credentials structure referenced by the *crp* parameter. The *which* parameter is one of the values of **PRIV_EFFECTIVE**, **PRIV_MAXIMUM**, **PRIV_INHERITED**, **PRIV_LIMITING**, and **PRIV_USED**. The corresponding privilege set is copied to the *privset* parameter. The *crp* parameter must be a valid, referenced credentials structure.

Execution Environment

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

Return Values

Item	Description
0	Success.
-1	An error has occurred.

Related information:

Security Kernel Services

kcred_getpriv Kernel Service

Purpose

Copies a privilege vector from a credentials structure.

Syntax

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

#include <sys/cred.h>

int kcred_getpriv ( crp, which, priv )
struct ucred * cr;
int which;
priv_t * priv;
```

Parameters

Item	Description
crp	Pointer to a credentials structure
which	Privilege set to get
priv	Privilege set

Description

The **kcred_getpriv** kernel service returns a single privilege set from the credentials structure referenced by *crp*. The *which* parameter is one of **PRIV_BEQUEATH**, **PRIV_EFFECTIVE**, **PRIV_INHERITED**, or **PRIV_MAXIMUM**. The corresponding privilege set will be copied to *priv*. *rp* must be a valid, referenced credentials structure.

Execution Environment

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

Return Values

Item	Description
0	Success. to priv.
-1	An error has occurred.

Related information:

Security Kernel Services

kcred_setcap Kernel Service

Purpose

Copies a capabilities set into a credentials structure.

Syntax

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

#include <sys/cred.h>

void kcred_setcap ( crp, cap )
struct ucred * cr;
struct __cap_t * cap;
```

Parameters

Item	Description
crp	Pointer to a credentials structure
cap	Capabilities set

Description

The **kcred_setcap** kernel service initializes the capability set in the credentials structure referenced by *crp* with *cap*. *rp* must be a valid, referenced credentials structure and must not be the current credentials of any process.

Execution Environment

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

Return Values

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

Related information:

Security Kernel Services

kcred_setgroups Kernel Service

Purpose

Copies a concurrent group set into a credentials structure.

Syntax

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

int kcred_setgroups ( crp, ngroups, groups )
struct ucred * cr;
int ngroups;
gid_t * groups;
```

Parameters

Item	Description
crp	Pointer to a credentials structure
ngroups	Size of the array of group ID values
groups	Array of group ID values

Description

The **kcred_setgroups** kernel service copies *ngroups* concurrent group set members into the credentials structure pointed to by *crp*. *crp* must be a valid, referenced credentials structure and must not be the current credentials of any process.

Execution Environment

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

Return Values

Item	Description
0	The concurrent group set has been copied successfully.
-1	An error has occurred.

Related information:

Security Kernel Services

kcred_setpag or kcred_setpag64 Kernel Service

Purpose

Copies a process authentication group ID into a credentials structure.

Syntax

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

int kcred_setpag ( crp, which, pag )
struct ucred * cr;
int which;
int pag;

int kcred_setpag64 ( crp, which, pag )
struct ucred * cr;
int which;
uint64 * pag;
```

Parameters

Item	Description
crp	Pointer to a credentials structure
which	PAG ID to set
pag	Process authentication group

Description

The **kcred_setpag** or **kcred_setpag64** kernel service initializes the specified PAG in the credentials structure referenced by *crp* with *pag*. The value of *which* must be a defined PAG ID. The PAG ID for the *Distributed Computing Environment* (DCE) is 0. *Crp* must be a valid, referenced credentials structure. *crp* may be a reference to the current credentials of a process.

Execution Environment

The **kcred_setpag** or **kcred_setpag64** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Success.
-1	An error has occurred.

Related information:

Security Kernel Services

kcred_setpagname Kernel Service

Purpose

Copies a process authentication group ID into a credentials structure.

Syntax

```
int kcred_setpagname (name, flags, func)
char *name;
int flags;
```

Description

The **kcred_setpagname** kernel service registers the name of a PAG and returns the PAG type identifier. If the PAG name has already been registered, the previously returned PAG type identifier is returned if the *flags* and *func* parameters match their earlier values.

Parameters

Item	Description
name	The *name* parameter is a 1 to 4 character, NULL-terminated name for the PAG type. Typical values might include "*afs*", "*dfs*", "*pki*" and "*krb5*."
flags	The *flags* parameter indicates if each PAG value is unique (PAG_UNIQUEVALUE) or multivalued (PAG_MULTIVALUED). A multivalued PAG type allows multiple calls to the **kcred_setpag** kernel service to be made to store multiple values for a single PAG type.
func	The *func* parameter is a pointer to an allocating and deallocating function. The *flag* parameter to that function is either PAGVALUE_ALLOC or PAGVALUE_FREE. The *value* parameter is the actual PAG value. The *func* parameter will be invoked by the **crfree** kernel service with a flag value of PAGVALUE_FREE on the last free value of a credential. Whenever a credentials structure is initialized with new PAG values, *func* will be invoked by that function with a value of PAGVALUE_ALLOC. This parameter may be ignored and an error returned if the value of *func* is non-NULL.

Return Values

A value of 0 or greater is returned upon successful completion. This value is the PAG type identifier which is used with other kernel services, such as the **kcred_getpag** and **kcred_setpag** subroutines . A negative value is returned if unsuccessful.

Error Codes

Item	Description
ENOSPC	The PAG table is full.
EEXISTS	The named PAG type already exists in the table and the *flags* and *func* parameters do not match their earlier values.
EINVAL	The *flags* parameter is an invalid value.

Related reference:

"__pag_setname System Call" on page 400

"__pag_setvalue System Call" on page 401

"kcred_getpagname Kernel Service" on page 246

kcred_setppriv Kernel Service

Purpose

Copies a privilege vector into a credentials structure.

Syntax

```
#include <sys/priv.h>
#include <sys/cred.h>

int kcred_setppriv (crp, which, privset)
struct ucred *crp;
int which;
privg_t privset;
```

Parameters

Item	Description
crp	Points to a credentials structure.
which	Specifies the privilege set to set.
privset	Specifies the privilege set.

Description

The **kcred_setppriv** kernel service sets one or more single privilege sets in the credentials structure referenced by the *crp* parameter. The *which* parameter is the bitwise OR of one or more values of **PRIV_EFFECTIVE**, **PRIV_MAXIMUM**, **PRIV_INHERITED**, **PRIV_LIMITING**, and **PRIV_USED**. The *privset* parameter initializes the corresponding privilege sets. The *crp* parameter must be a valid, referenced credentials structure and cannot be the current credentials of any process.

Execution Environment

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

Return Values

Item	Description
0	Success.
-1	An error has occurred.

Related information:

Security Kernel Services

kcred_setpriv Kernel Service

Purpose

Copies a privilege vector into a credentials structure.

Syntax

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

#include <sys/cred.h>

int kcred_setpriv ( crp, which, priv )
struct ucred * cr;
int which;
priv_t * priv;
```

Parameters

Item	Description
crp	Pointer to a credentials structure
which	Privilege set to set
priv	Privilege set

Description

The **kcred_setpriv** kernel service sets one or more single privilege sets in the credentials structure referenced by *crp*. The *which* parameter is one or more bit-wise ored values of **PRIV_BEQUEATH**, **PRIV_EFFECTIVE**, **PRIV_INHERITED**, and **PRIV_MAXIMUM**. The corresponding privilege sets are initialized from *priv*. *crp* must be a valid, referenced credentials structure and must not be the current credentials of any process.

Execution Environment

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

Return Values

Item	Description
0	Success. to priv.
-1	An error has occurred.

Related information:

Security Kernel Services

kern_soaccept Kernel Service

Purpose

Accepts the first queued connection by assigning it to the new socket.

Syntax

```
#include <sys/kern_socket.h>
int kern_soaccept( ksocket_t  so,
ksocket_t  *aso,
struct mbuf **name,
int nonblock )
```

Parameters

Item	Description
so	The socket that is used in the **kern_solisten()** Kernel Service.
aso	The new socket for the accepted connection. The caller must pass in the address of the **ksocket_t**.
name	A **struct sockadr** address is returned in a **mbuf** buffer whose address is stored in the ***name** parameter. The caller should pass in the address of the struct **mbuf** * structure. The caller sets the **mbuf** buffer free after the function returns successfully.
nonblock	A flag to specify if this call should be nonblocking. The value of 1 is for nonblocking and 0 is for blocking.

Description

The **kern_soaccept** kernel service accepts the first queued connection by assigning it to the new socket.

Execution Environment

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

Examples

```
struct mbuf *name = NULL;
ksocket_t   so;
ksocket_t   aso;
struct sockaddr_in laddr;
int       rc;
rc = kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
if (rc != 0 )
{
      return(-1);
}
bzero(&laddr, sizeof(struct sockaddr_in));
laddr.sin_family = AF_INET;
laddr.sin_port = 12345;
laddr.sin_len = sizeof(struct sockaddr_in);
laddr.sin_addr.s_addr = inet_addr("9.3.108.208");
rc = kern_sobind(so, (struct sockaddr *)&laddr);
if (rc != 0 )
{
     return(-1);
}
rc = kern_solisten(so, 5);
if (rc != 0 )
{
     return(-1);
}
rc = kern_soaccept(so, &aso, &name, 0);
if (rc != 0 )
{
     return(-1);
}
m_freem(name); /* Caller needs to free the mbuf after kern_soaccept */
```

Return Values

Item	Description
0	Upon Success
>0	Error

The non-zero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"kern_socreate Kernel Service" on page 257

"kern_soreceive Kernel Service" on page 260

"kern_sosend Kernel Service" on page 263

kern_sobind Kernel Service

Purpose

Associates the local network address to the socket.

Syntax

```
#include <sys/kern_socket.h>
int  kern_sobind( ksocket_t  so, struct sockaddr *laddr )
```

Parameters

Item	Description
so	The socket that was created by the **kern_socreate()** system call.
laddr	Local address to be bound.

Description

The **kern_sobind** kernel service binds a local address to the socket.

Execution Environment

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

Examples

```
ksocket_t   so;
struct sockaddr_in laddr;
int       rc;
rc = kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
if (rc != 0 )
{
      return(-1);
}
bzero(&laddr, sizeof(struct sockaddr_in));
laddr.sin_family = AF_INET;
laddr.sin_port = 12345;
laddr.sin_len = sizeof(struct sockaddr_in);
laddr.sin_addr.s_addr = inet_addr("9.3.108.208");
rc = kern_sobind(so, (struct sockaddr *) &laddr);
if (rc != 0 )
{
      return(-1);
}
```

Return Values

Item	Description
0	Upon Success
>0	Error

The nonzero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"kern_socreate Kernel Service" on page 257

"kern_solisten Kernel Service" on page 259

kern_soclose Kernel Service

Purpose

Aborts any connections and releases the data in the socket.

Syntax

```
#include <sys/kern_socket.h>
int  kern_soclose( ksocket _t  so )
```

Parameters

Item	Description
so	The socket on which the close will be issued.

Description

The **kern_soclose** kernel service aborts any connection and releases the data in the socket.

Execution Environment

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

Examples

```
ksocket_t   so;
int       rc;
rc = kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
if (rc != 0 )
{
    return(-1);
}
/* Socket is in use */
...
kern_soclose(so);
```

Return Values

Item	Description
0	Upon Success
>0	Error

The nonzero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"kern_socreate Kernel Service" on page 257

kern_soconnect Kernel Service

Purpose

Establishes a connection to a foreign address.

Syntax

```
#include <sys/kern_socket.h>
int kern_soconnect( ksocket_t  so, struct sockaddr *faddr )
```

Parameters

Item	Description
so	The socket that was created by **socreate()**.
faddr	Foreign address to connect.

Description

The **kern_soconnect** kernel service establishes connection with a foreign address.

Execution Environment

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

Examples

```
ksocket_t   so;
struct sockaddr_in faddr;
int       rc;
rc = kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
if (rc != 0 )
{
   return(-1);
}
bzero(&faddr, sizeof(struct sockaddr_in));
faddr.sin_family = AF_INET;
faddr.sin_port = 23456;
faddr.sin_len = sizeof(struct sockaddr_in);
faddr.sin_addr.s_addr = inet_addr("9.3.108.210");
rc = kern_soconnect(so, (struct sockaddr *) &faddr);
if (rc != 0 )
{
   return(-1);
}
```

Return Values

Item	Description
0	Upon Success
>0	Error

The nonzero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"kern_socreate Kernel Service"

"kern_sosend Kernel Service" on page 263

"kern_soreceive Kernel Service" on page 260

kern_socreate Kernel Service

Purpose

Used to create a socket of the specified address family and type. If the protocol is left unspecified (zero), then the system selects the protocol based on the address family and type.

Syntax

```
#include <sys/kern_socket.h>
int kern_socreate (int addressfamily, ksocket_t  *so, int type, int protocol)
```

Parameters

Item	Description
addressfamily	The address family for the newly created socket. The file **<sys/socket.h>** contains the definitions for the family. Currently AIX supports: **AF_INET** Denotes the IPv4 Internet addresses. **AF_INET6** Denotes the IPv6 Internet addresses.
so	The socket assigned by the **create()** call. The caller must pass the address of **ksocket_t**.
type	The requested socket type. The file **<sys/socket.h>** contains the definition for the socket type. Currently AIX supports **SOCK_STREAM**.
protocol	The file **<netinet/in.h>** contains the definition for the protocol. Currently AIX supports **IPPROTO_TCP**

Description

The **kern_socreate** kernel service creates a socket based on the address family, type and protocol.

Execution Environment

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

Examples

```
ksocket_t      so;
ksocket _t     so2;
kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
kern_socreate(AF_INET6, &so2, SOCK_STREAM, 0);
```

Return Values

Item	Description
0	Upon Success
>0	Error

The nonzero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:
"kern_soclose Kernel Service" on page 255
"kern_soconnect Kernel Service" on page 256
"kern_soshutdown Kernel Service" on page 265

kern_sogetopt Kernel Service

Purpose

Obtains the option associated with the socket, either at the socket level or at the protocol level.

Syntax

```
#include <sys/kern_socket.h>
int  kern_sogetopt( ksocket_t  so, int level, int optname, struct mbuf **mp )
```

Parameters

Item	Description
so	The socket that will be used to retrieve the option.
level	The socket level (e.g. **SOL_SOCKET**) or protocol level (**IPPROTO_TCP**)
optname	The option name to retrieve. Socket options can be found in **<sys/socket.h>** and TCP options can be found in **<netinet/tcp.h>** mp
mp	The **mbuf** that will be returned with the option value. The **mp->m_len** will be the size of the value. The caller must pass the address of the **struct mbuf** *. The caller must set the **mbuf** free after the function returns successfully.

Description

The **kern_sogetopt** kernel service obtains the option associated with the socket, either at the socket level, or at the protocol level.

Execution Environment

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

Examples

```
ksocket_t   so;
int       rc;
struct mbuf *sopt = NULL;
int  tcp_nodelay = -1;
rc = kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
if (rc != 0 )
{
   return(-1);
}
rc = sogetopt(so, IPPROTO_TCP, TCP_NODELAY, &sopt);
if (rc != 0 )
{
   return(-1);
}
tcp_nodelay = *mtod(sopt, int *) ? 1 : 0;
m_free(sopt); /* Caller needs to free the mbuf after kern_sogetopt */
```

Return Values

Item	Description
0	Upon Success
>0	Error

The nonzero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"kern_socreate Kernel Service" on page 257

kern_solisten Kernel Service

Purpose

Prepares to accept incoming connections on the socket.

Syntax

```
#include <sys/kern_socket.h>
int kern_solisten( ksocket_t  so,  int backlog )
```

Parameters

Item	Description
so	The socket that was created by **kern_socreate()** and used in **kern_sobind()**
backlog	Limit the number of connection requests that can be queued on this socket. The maximum value that can be passed to this parameter equals the minimum number of user backlog number and the network option **somaxconn** value.

Description

The **kern_solisten** kernel service prepares to accept incoming connection on the socket.

Execution Environment

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

Examples

```
struct mbuf *name = NULL;
ksocket_t   so;
struct sockaddr_in laddr;
int       rc;
```

```
rc = kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
if (rc != 0 )
{
    return(-1);
}
bzero(&laddr, sizeof(struct sockaddr_in));
laddr.sin_family = AF_INET;
laddr.sin_port = 12345;
laddr.sin_len = sizeof(struct sockaddr_in);
laddr.sin_addr.s_addr = inet_addr("9.3.108.208");
rc = kern_sobind(so, (struct sockaddr *)&laddr);
if (rc != 0 )
{
    return(-1);
}
rc = kern_solisten(so, 5);
if (rc != 0 )
{
    return(-1);
}
```

Return Values

Item	Description
0	Upon Success
>0	Error

The nonzero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"kern_socreate Kernel Service" on page 257

"kern_soaccept Kernel Service" on page 253

kern_soreceive Kernel Service

Purpose

The routine processes one record per call and returns the number of bytes requested.

Syntax

```
#include <sys/kern_socket.h>
int  kern_soreceive( ksocket_t   so,
struct mbuf **paddr,
long len,
struct mbuf **mp,
struct mbuf **controlp,
int *flagp )
```

Parameters

Item	Description
so	The socket to receive the data.
paddr	The foreign socket address information is returned in this pointer. Caller should pass in address of **struct mbuf** * . Caller needs to free this **mbuf** after the function returns successfully. Caller can pass in NULL if caller doesn't need foreign address information.
len	The length of the data to be received.
mp	The **mbuf** pointer so that data can be returned in an **mbuf** chain. The caller must pass in the address of **struct mbuf ***. The caller must free this **mbuf** after the function returns.
controlp	Pointer to an **mbuf** containing the control information. Caller should pass in address of **struct mbuf *** . Caller needs to free this **mbuf** after the function returns successfully. Caller can pass in NULL if there is no control information.

Item	Description
flagp	If *flagp* is not NULL, caller can pass in actual flag. The flags are defined in the **<sys/socket.h>** file. The **kern_soreceive** routine will use flags set in *flagp*. The caller can set the *flagsp* to **MSG_WAITALL** or **MSG_NONBLOCK**. On return, it will set *flagp* to **MSG_TRUNC**, **MSG_OOB** wherever applicable.

Description

The **kern_soreceive** kernel service processes one record per call and returns the number of bytes that are requested. If there is data in the socket receive buffer, the **kern_soreceive** kernel service returns up to < len> bytes as a **mbuf** chain. The actual number of bytes returned is computed by adding the **m_len** fields of each **mbuf** in the chain. If there is no data, but the connection is still established, the **kern_soreceive** kernel service either returns EAGAIN with *mp set to NULL (if MSG_NONBLOCK is set) or returns wait for data to arrive (if MSG_NONBLOCK is not set). If the connection is closed before the call or while waiting for data, the *mp is set to NULL and 0 is returned. Waiting may be interrupted, in which case **kern_soreceive** returns EAGAIN, EINTR, or ERESTART and *mp is undefined. The application might return EINTR, but calls the **kern_soreceive** kernel service again.

Execution Environment

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

Examples

```
ksocket_t   so;
struct mbuf *data = NULL;
struct sockaddr_in faddr;
long        len = 512;
int         flags = 0;
int         rc;
rc = kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
if (rc != 0 )
{
   return(-1);
}
bzero(&faddr, sizeof(struct sockaddr_in));
faddr.sin_family = AF_INET;
faddr.sin_port = 23456;
faddr.sin_len = sizeof(struct sockaddr_in);
faddr.sin_addr.s_addr = inet_addr("9.3.108.210");
rc = kern_soconnect(so, (struct sockaddr *) &faddr);
if (rc != 0 )
{
   return(-1);
}
do
{
   rc = kern_soreceive(so, NULL, len, &data, NULL, &flags);
}  while (rc == EAGAIN || rc == EINTR || rc == ERESTART);
if ((rc == 0) && data)
{
  /* process the data */
  ...
  m_freem(data); /* Caller needs to free the mbuf after kern_soreceive. */
}
else
{
   return(-1);
}
```

Return Values

Item	Description
0	Upon Success
>0	Error

The nonzero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"kern_socreate Kernel Service" on page 257

"kern_sosend Kernel Service" on page 263

kern_soreserve Kernel Service

Purpose

The routine enforces the limit for the send and receive buffer space for a socket. It does not actually allocate memory only sets the buffer size.

Syntax

```
#include <sys/kern_socket.h>
int kern_soreserve( ksocket_t  so, uint64_t sndcc, uint64_t rcvcc )
```

Parameters

Item	Description
so	The socket that will be used in reserving the space.
sndcc	Send buffer size in bytes.
rcvcc	Receive buffer size in bytes.

Description

The **kern_soreserve** kernel service enforces the limit for the send and receive buffer space for a socket. It does not actually allocate memory. It sets the buffer size.

Execution Environment

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

Examples

```
ksocket_t   so;
uint64_t    sb_snd_hiwat = 2048;
uint64_t    sb_rcv_hiwat = 2048;
int         rc;
rc = kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
if (rc != 0 )
{
   return(-1);
}
rc = kern_soreserve(so, sb_snd_hiwat, sb_rcv_hiwat);
if (rc != 0 )
{
     return(-1);
}
```

Return Values

Item	Description
0	Upon Success
>0	Error

The nonzero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"kern_socreate Kernel Service" on page 257

kern_sosend Kernel Service

Purpose

Pass data and control information to the protocol associated send routines.

Syntax

```
#include <sys/kern_socket.h>
int  kern_sosend( ksocket_t  so, struct sockaddr *faddr,
struct mbuf *top,
struct mbuf *control,
int flags )
```

Parameters

Item	Description
so	The socket to send data.
faddr	The destination address, only necessary if the socket is not connected.
top	The **mbuf** chain of data to be sent. Remember that the first **mbuf** must have the packet header filled out. Set the **top->m_pkthdr.len** to the total length of the data in the **mbuf** chain and the **m_flags** to **M_PKTHDR**. The caller must allocate **mbuf** memory before calling the routine.
control	Pointer to an **mbuf** containing the control information to be sent. The caller must allocate **mbuf** memory before calling the function if the caller wants to pass in control information.
flags	Flags options for this write call. Caller can set flags to **MSG_NONBLOCK**.

Description

The **kern_sosend** kernel service passes data and control information to the protocol associated send routines.

Execution Environment

The **kern_sosend** kernel service can be called from process environment.

Examples

```
ksocket_t  so;
int flags = 0;
struct sockaddr_in faddr;
struct mbuf *send_mbuf;
struct sockaddr_in  faddr;
char   msg[100];
int     i, rc;
rc = kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
if (rc != 0 )
{
    return(-1);
}
bzero(&faddr, sizeof(struct sockaddr_in));
faddr.sin_family = AF_INET;
faddr.sin_port = 23456;
faddr.sin_len = sizeof(struct sockaddr_in);
faddr.sin_addr.s_addr = inet_addr("9.3.108.210");
rc = kern_soconnect(so, (struct sockaddr *) &faddr);
if (rc != 0 )
{
      return(-1);
}
send_mbuf = MGETBUF(sizeof(msg), M_DONOTWAIT); /* Caller needs to allocate mbuf memory */
if (send_mbuf == NULL)
{
     return (-1);
}
for (i=0; i < 100, i++)
{
        msg[i] = 0x2A;
}
bcopy(msg, mtod(send_mbuf, caddr_t), sizeof(msg));
send_mbuf->m_len = send_mbuf->m_pkthdr.len = sizeof(msg);
rc = kern_sosend(so, NULL, send_mbuf, 0, MSG_NONBLOCK));
if (rc != 0 )
{
     return(-1);
}
```

Return Values

Item	Description
0	Upon Success
>0	Error

The nonzero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:

“kern_socreate Kernel Service” on page 257

“kern_soreceive Kernel Service” on page 260

kern_sosetopt Kernel Service

Purpose

Sets the option associated with the socket, either at the socket level or at the protocol level.

Syntax

```
#include <sys/kern_socket.h>
int  sosetopt( ksocket_t  so,
int level, int optname,
struct mbuf *mp )
```

Parameters

Item	Description
so	The socket that will be used to set the option.
level	The socket level (e.g. **SOL_SOCKET**) or protocol level (**IPPROTO_TCP**)
optname	The option name to set. Socket options can be found in **<sys/socket.h>** and the TCP options can be found in **<netinet/tcp.h>**.
mp	The **mbuf** that contains the option value and will be used to modify the field specified by the option name. The **mp->m_len** should be the size of the value. The caller must allocate **mbuf** memory before calling the function.

Description

The **kern_sosettopt** kernel service sets the option associated with the socket, either at the socket level, or at the protocol level.

Execution Environment

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

Examples

```
ksocket_t    so;
struct mbuf *mp = NULL;
struct linger *linger;
int rc;
rc = kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);
if (rc != 0 )
{
       return(-1);
}
mp = m_get(M_DONTWAIT, MT_SOOPTS); /* Caller of kern_sosetopt needs to allocated mbuf  memory */
if (mp == NULL)
{
       return (-1);
}
mp->m_len = sizeof(struct linger);
linger = mtod(mp, struct linger *);
linger->l_linger = 5;
linger->l_onoff = 1;
rc = kern_sosetopt(so, SOL_SOCKET, SO_LINGER, mp);
if (rc != 0 )
{
       return(-1);
}
```

Return Values

Item	Description
0	Upon Success
>0	Error

The nonzero return value is the error number that is defined in the **/usr/include/sys/errno.h** file.

Related reference:

"kern_socreate Kernel Service" on page 257

kern_soshutdown Kernel Service

Purpose

Closes the read-half, write-half or both read and write of a connection.

Syntax

```
#include <sys/kern_socket.h>
int  kern_soshutdown( ksocket_t  so, int how )
```

Parameters

Item	Description
so	The socket to which the shutdown will be issued.
how	0 read, 1 write, 2 read and write

Description

The **kern_soshutdown** kernel service closes the read-half, write-half or both read and write of a connection.

Execution Environment

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

Examples

```
ksocket_t    so;

/* Create the socket so */

kern_socreate(AF_INET, &so, SOCK_STREAM, IPPROTO_TCP);

/* Shutting down both the read/write */

kern_soshutdown(so, 2);
```

Return Values

Item	Description
0	Upon Success
>0	Error

Related reference:

"kern_socreate Kernel Service" on page 257

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

Item	Description
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

Item	Description
0	Indicates successful completion.

Related information:

sethostname subroutine

Network Kernel Services

kgetpname Kernel Service

Purpose

Provides the calling process's base program name.

Syntax

```
#include <sys/encap.h>
int kgetpname (char * Buffer, size_t *BufferSize);
```

Description

The **kgetpname** kernel service copies the program name of the calling process into the buffer specified by *Buffer*. Including the null terminator, the service copies no more than the lesser of **BufferSize*, **MAXCOMLEN**, or the actual size of the program name in bytes into the buffer. If *Buffer* is NULL, or **BufferSize* is `0`, no copy is performed. If the full program name is copied into the buffer, the total number of bytes copied is written to **BufferSize*. If **kgetpname** cannot copy the full program name into the buffer, the size in bytes of the full program name is written to **BufferSize*, and **ENAMETOOLONG** is returned.

Execution Environment

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

Return Values

Item	Description
0	The full program name was successfully written to the buffer.
ENAMETOOLONG	Only part of the full program name was written to the buffer, and **kgetpname** stored the (positive) length in bytes (including the null character) of the full program name into **BufferSize*.
EINVAL	*Buffer* is Null, *BufferSize* is NULL, or **BufferSize* is 0.
ENOTSUP	The **kgetpname** kernel service was called from inside an interrupt context.

kgetrlimit64 Kernel Service

Purpose

Controls maximum system resource consumption.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>
#include <sys/resource.h>

void kgetrlimit64 (Resource1, RLP)
int Resource1;
struct rlimit64 *RLP;
```

Parameters

Item **Description**

Resource1 The *Resource1* parameter can be one of the following values:

RLIMIT_AS
The maximum size of a process's total available memory, in bytes. This limit is not enforced.

RLIMIT_CORE
The largest size, in bytes, of a core file that can be created. This limit is enforced by the kernel. If the value of the RLIMIT_FSIZE limit is less than the value of the RLIMIT_CORE limit, the system uses the RLIMIT_FSIZE limit value as the soft limit.

RLIMIT_CPU
The maximum amount of central processing unit (CPU) time, in seconds, to be used by each process. If a process exceeds its soft CPU limit, the kernel sends a SIGXCPU signal to the process. After the hard limit is reached, the process is killed with SIGXCPU, even if it handles, blocks, or ignores that signal.

RLIMIT_DATA
The maximum size, in bytes, of the data region for a process. This limit defines how far a program can extend its break value with the **sbrk** subroutine. This limit is enforced by the kernel.

RLIMIT_FSIZE
The largest size, in bytes, of any single file that can be created. When a process attempts to write, truncate, or clear beyond its soft RLIMIT_FSIZE limit, the operation fails with the **errno** variable set to EFBIG. If the environment variable XPG_SUS_ENV=ON is set in the user's environment before the process is issued, then the SIGXFSZ signal is also generated.

RLIMIT_NOFILE
This is a number one greater than the maximum value that the system can assign to a newly-created descriptor.

RLIMIT_STACK
The maximum size, in bytes, of the stack region for a process. This limit defines how far a program stack region can be extended. The system automatically performs stack extension. This limit is enforced by the kernel. When the stack limit is reached, the process receives a SIGSEGV signal. If this signal is not caught by a handler using the signal stack, the signal ends the process.

RLIMIT_RSS
The maximum size, in bytes, to which the resident set size of a process can grow. This limit is not enforced by the kernel. A process might exceed its soft limit size without being ended.

RLP Points to the **rlimit64** structure where the requested limits are returned by the **kgetrlimit64** kernel service.

Description

The **kgetrlimit64** kernel service returns the values of limits on system resources used by the current process and its children processes.

Note: The initial values returned by the **kgetrlimit64** kernel service are the ulimit values in effect when the process was started. For maxdata programs the initial soft limit for data is set to the lower of data ulimit value or a value corresponding to the number of data segments reserved for data segments.

The **rlimit64** structure specifies the hard and soft limits for a resource, as defined in the **sys/resource.h** file. The RLIM64_INFINITY value defines an infinite value for a limit.

Execution Environment

The **kgetrlimit64** kernel service can be called from either the process or interrupt environment.

Return Values

The **kgetrlimit64** kernel service has no return values.

Related information:

getrlimit64 subroutine

kgetsystemcfg Kernel Service

Purpose

Displays the system configuration information.

Syntax

```
#include <systemcfg.h>
uint64_t kgetsystemcfg ( int name)
```

Description

Displays the system configuration information.

Parameters

Item	Description
name	Specifies the system variable setting to be returned. Valid values for the *name* parameter are defined in the **systemcfg.h** file.

Return Values

If the value specified by the *name* parameter is system-defined, the **kgetsystemcfg** kernel service returns the data that is associated with the structure member represented by the *input* parameter. Otherwise, the **kgetsystemcfg** kernel service will return **UINT64_MAX**, and the **errno** will be set.

Error Codes

The **kgetsystemcfg** subroutine will fail if:

Item	Description
EINVAL	The value of the *name* parameter is invalid.

Related information:

getsystemcfg subroutine

kgettickd Kernel Service

Purpose

Retrieves the current status of the systemwide time-of-day timer-adjustment values.

Syntax

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

int kgettickd (timed, tickd, time_adjusted)
int *timed;
int *tickd;
int *time_adjusted;
```

Parameters

Item	Description
timed	Specifies the current amount of time adjustment in microseconds remaining to be applied to the systemwide timer.
tickd	Specifies the time-adjustment rate in microseconds.
time_adjusted	Indicates if the systemwide timer has been adjusted. A value of True indicates that the timer has been adjusted by a call to the **adjtime** or **settimer** subroutine. A value of False indicates that it has not. The use of the **ksettimer** kernel service has no effect on this flag. This flag can be changed by the **ksettickd** kernel service.

Description

The **kgettickd** kernel service provides kernel extensions with the capability to determine if the **adjtime** or **settimer** subroutine has adjusted or changed the systemwide timer.

The **kgettickd** kernel service is typically used only by kernel extensions providing time synchronization functions. This includes coordinated network time (which is the periodic synchronization of all system clocks to a common time by a time server or set of time servers on a network), where use of the **adjtime** subroutine is insufficient.

Execution Environment

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

Return Values

The **kgettickd** service always returns a value of 0.

Related reference:

"ksettimer Kernel Service" on page 308

Related information:

Timer and Time-of-Day Kernel Services

Using Fine Granularity Timer Services and Structures

kkey_assign_private Kernel Service

Purpose

Requests a private kernel-key assignment.

Syntax

```
#include <sys/types.h>
#include <sys/skeys.h>
#include <sys/kerrno.h>

kerrno_t kkey_assign_private (id, instance, flags, kkey)
char *id;
long instance;
unsigned long flags;
kkey_t *kkey;
```

Parameters

Item	Description
id	Specifies a null-terminated string. The **kkey_assign_private** kernel service uses the string value to assign a private key. This normally contains a load module name associated with the calling kernel subsystem, but you can specify any unique string.
instance	Specifies a unique number for each private key requested by a subsystem. This must be an integer value starting from 0 and increases with each kernel-key requested.
flags	You must specify this parameter to zero.
kkey	Contains the returned assigned kernel key. The valid pointer must be a 4-byte aligned address (**kkey_t**'s natural alignment).

Description

The **kkey_assign_private** kernel service assigns a private kernel key to the caller. Private kernel keys are used to limit data accessibility by external kernel code. The **kkey_assign_private** kernel service distributes requests for private kernel keys among a predetermined range (from KKEY_PRIVATE1 to KKEY_PRIVATE32). The intention is to perform a uniform distribution on behalf of requests by multiple kernel subsystems. The assignment is made based on the *id* and *instance* parameters and might return the same private key to multiple callers. It might also return the same private key when the instance number is different.

The **kkey_assign_private** kernel service does not perform a resource allocation. It only provides a recommended kernel key to use for data protection.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_KKEY_ASSIGN_PRIVATE	Indicates that the parameter or execution environment is not valid.

kkeyset_add_key Kernel Service

Purpose

Adds a kernel key to a kernel keyset.

Syntax

```
#include <sys/kerrno.h>
#include <sys/skeys.h>

kerrno_t kkeyset_add_key (set, key, flags)
kkeyset_t set;
kkey_t key;
unsigned long flags;
```

Parameters

Item	Description
set	Specifies the kernel keyset to which the **kkeyset_add_key** kernel service will add a key.
key	Specifies the kernel key to add.
flags	You can specify the *flags* parameter to one of the following values: **KA_READ** Specifies that the read access for the key is to be added. **KA_WRITE** Specifies that the write access for the key is to be added. **KA_RW** Specifies that both the read access and the write access are to be added. This is equivalent to the value of **KA_READ** | **KA_WRITE**.

Description

The **kkeyset_add_key** kernel service adds a single kernel key specified by the *key* parameter to the kernel keyset specified by the *set* parameter. You must specify the *flags* parameter to control the read or write authority.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_KKEYSET_ADD_KEY	Indicates that the parameter or execution environment is not valid.

kkeyset_add_set Kernel Service

Purpose

Adds members of one kernel keyset to an existing kernel keyset.

Syntax

```
#include <sys/kerrno.h>
#include <sys/skeys.h>

kerrno_t kkeyset_add_set (set, addset)
kkeyset_t set;
kkeyset_t addset;
```

Parameters

Item	Description
set	Specifies an existing kernel keyset. This set contains the resulting union on completion.
addset	Specifies the kernel keyset to add.

Description

The **kkeyset_add_set** kernel service adds a kernel keyset specified by the *addset* parameter to the kernel keyset specified by the *set* parameter.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_KKEYSET_ADD_SET	Indicates that the parameter or execution environment is not valid.

kkeyset_create Kernel Service

Purpose

Creates and initializes a kernel keyset.

Syntax

```
#include <sys/kerrno.h>
#include <sys/skeys.h>

kerrno_t kkeyset_create (set)
kkeyset_t *set;
```

Parameters

Item	Description
set	Contains the returned newly-created keyset.

Description

The **kkeyset_create** kernel service creates a new (empty) kernel keyset. You can add or remove the access to an individual or groups of kernel keys using the **kkeyset_add_key**, **kkeyset_remove_key**, **kkeyset_add_set**, and **kkeyset_remove_set** kernel services.

Important: The **kkeyset_create** kernel service allocates hidden kernel resources. You must release these resources using the **kkeyset_delete** kernel service when the kernel keyset is no longer in use. When creating a new set, the caller of the **kkeyset_create** kernel service must initialize the storage that will contain the returned kernel keyset (**set*) to the value of KKEYSET_INVALID.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
ENOMEM_KKEYSET_CREATE	Indicates that the available memory is not sufficient to satisfy the request.
EINVAL_KKEYSET_CREATE	Indicates that the parameter or execution environment is not valid.

Related reference:

"kkeyset_add_key Kernel Service" on page 272

"kkeyset_remove_key Kernel Service" on page 275

"kkeyset_delete Kernel Service"

kkeyset_delete Kernel Service

Purpose

Deletes a kernel keyset.

Syntax

```
#include <sys/kerrno.h>
#include <sys/skeys.h>

kerrno_t kkeyset_delete (set)
kkeyset_t set;
```

Parameters

Item	Description
set	Specifies the keyset to be destroyed.

Description

The **kkeyset_delete** kernel service destroys a kernel keyset. The kernel service releases the hidden resources associated with this keyset.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_KKEYSET_DELETE	Indicates that the parameter or execution environment is not valid.

kkeyset_remove_key Kernel Service

Purpose

Removes a kernel key from a kernel keyset.

Syntax

```
#include <sys/kerrno.h>
#include <sys/skeys.h>

kerrno_t kkeyset_remove_key (set, key, flags)
kkeyset_t set;
kkey_t key;
unsigned long flags;
```

Parameters

Item	Description
set	Specifies the kernel keyset from which the **kkeyset_remove_key** kernel service will remove a key.
key	Specifies the kernel key to remove.
flags	You can specify the *flags* parameter to one of the following values: **KA_READ** Specifies that the read access for the key is to be removed. **KA_WRITE** Specifies that the write access for the key is to be removed. **KA_RW** Specifies that both the read access and the write access are to be removed. This is equivalent to the value of **KA_READ** | **KA_WRITE**.

Description

The **kkeyset_remove_key** kernel service removes a single kernel key specified by the *key* parameter from the kernel keyset specified by the *set* parameter. You must specify the *flags* parameter to control the read or write authority.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_KKEYSET_REMOVE_KEY	Indicates that the parameter or execution environment is not valid.

kkeyset_remove_set Kernel Service

Purpose

Removes members of one kernel keyset from an existing kernel keyset.

Syntax

```
#include <sys/kerrno.h>
#include <sys/skeys.h>

kerrno_t kkeyset_remove_set (set, removeset)
kkeyset_t set;
kkeyset_t removeset;
```

Parameters

Item	Description
set	Specifies the kernel keyset from which the **kkeyset_remove_set** kernel service will remove a keyset.
removeset	Specifies the kernel keyset to remove.

Description

The **kkeyset_remove_set** kernel service removes a kernel keyset specified by the *removeset* parameter from the kernel keyset specified by the *set* parameter.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_KKEYSET_REMOVE_SET	Indicates that the parameter or execution environment is not valid.

kkeyset_to_hkeyset Kernel Service

Purpose

Computes the hardware keyset associated with a kernel keyset.

Syntax

```
#include <sys/kerrno.h>
#include <sys/skeys.h>

kerrno_t kkeyset_to_hkeyset (kkeyset, hkeyset)
kkeyset_t kkeyset;
hkeyset_t *hkeyset;
```

Parameters

Item	Description
kkeyset	Specifies the input kernel keyset to be mapped.
hkeyset	Specifies the hardware keyset that is mapped to. The valid pointer must be an 8-byte aligned address.

Description

The **kkeyset_to_hkeyset** kernel service maps a kernel keyset to its associated hardware keyset.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_KKEYSET_TO_HKEYSET	Indicates that the parameter or execution environment is not valid.

klpar_get_info Kernel Service

Purpose

Retrieves the calling partition's characteristics.

Syntax

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

int klpar_get_info (command, lparinfo, bufsize)
int command;
void *lparinfo;
size_t bufsize;
```

Parameters

Item	Description
command	Specifies whether the user wants **format1**, **format2**, or **processor module** details.
lparinfo	Pointer to the user-allocated buffer that is passed in.
bufsize	Size of the buffer that is passed in.

Description

The **klpar_get_info** kernel service retrieves LPAR and Micro-Partitioning attributes of both low-frequency use and high-frequency use and also retrieves processor module information. Because the low-frequency attributes, as defined in the **lpar_info_format1_t** structure, are static in nature, a reboot is required to effect any change. The high-frequncy attributes, as defined in the **lpar_info_format2_t** structure, can be changed dynamically while the partition is running. The signature of this kernel service, its parameter types, and the order of the member fields in both the **lpar_info_format1_t** and **lpar_info_format2_t** structures are specific to the AIX platform. If you requests **processor module** information, the kernel service provides this information as an array of **proc_module_info_t** structures. To obtain this information, the caller must provide a buffer of the exact length to accommodate one **proc_module_info_t** structure for each module type. You can obtain the module count using the **NUM_PROC_MODULE_TYPES** command. The module count is in the form of a **uint64_t** type. Processor module information is reported for the entire system. This information is available on POWER6® and later systems.

To see the complete structures of **lpar_info_format1_t**, **lpar_info_format2_t**, and **proc_module_info_t**, refer to the **dr.h** header file.

Return Values

Upon success, the **klpar_get_info** kernel service returns a value of 0. Upon failure, the **klpar_get_info** kernel service returns an error code.

Error Codes

Item	Description
EINVAL	Invalid input parameter.
ENOSYS	The hardware or the firmware level does not support this operation.
ENOTSUP	The platform does not support this operation.

Related information:

lpar_get_info subroutine

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

Item	Description
kmid	Specifies the kernel module ID of the object file for which the entry point is requested. This parameter is the kernel module ID returned by the **kmod_load** kernel service.
flags	Flag specifying entry point options. The following flag is defined: **0** Returns a function pointer to the specified module's entry point as specified in the module header.

Description

The **kmod_entrypt** kernel service obtains a function pointer to a specified module's entry point. This function pointer is typically used to invoke a routine in the module for initializing or terminating its functions. Initialization and termination occurs after loading and before unloading. The module for which the entry point is requested is specified by the kernel module ID represented by the *kmid* parameter.

Execution Environment

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

Return Values

A nonnull function pointer indicates a successful completion. This function pointer contains the module's entry point. A null function pointer indicates an error.

Related reference:

"kmod_load Kernel Service"

Related information:

Kernel Extension and Device Driver Management Kernel Services

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

Item **Description**

pathp
Points to a character string containing the path-name of the object file to load or query.

flags
Specifies a set of loader flags describing which loader options to invoke. The following flags are defined:

LD_USRPATH
The character strings pointed to by the *pathp* and *libpathp* parameters are in user address space. If the **LD_USRPATH** flag is not set, the character strings are assumed to be in kernel, or system, space.

LD_KERNELEX
Puts this object file's exported symbols into the **/usr/lib/boot/unix** name space. Additional object files loaded due to symbol resolution for the specified file do not have their exported symbols placed in kernel name space.

LD_SINGLELOAD
When this flag is set, the object file specified by the *pathp* parameter is loaded into the kernel only if an object file with the same path-name has not already been loaded. If an object file with the same path-name has already been loaded, its module ID is returned (using the *kmidp* parameter) and its load count incremented. If the object file is not yet loaded, this service performs the load as if the flag were not set.

This option is useful in supporting global kernel routines where only one copy of the routine and its data can be present. Typically, routines that export symbols to be added to kernel name space are of this type.

Note: A path-name comparison is done to determine whether the same object file has already been loaded. This service will erroneously load a new copy of the object file into the kernel if the path-name to the object file is expressed differently than it was on a previous load request.

If neither this flag nor the **LD_QUERY** flag is set, this service loads a new copy of the object file into the kernel. This occurs even if other copies of the object file have previously been loaded.

LD_QUERY
This flag specifies that a query operation will determine if the object file specified by the *pathp* parameter is loaded. If not loaded, a kernel module ID of 0 is returned using the *kmidp* parameter. Otherwise, the kernel module ID assigned to the object file is returned.

If multiple instances of this file have been loaded into the kernel, the kernel module ID of the most recently loaded object file is returned.

The *libpathp* parameter is not used for this option.

Note: A path-name comparison is done to determine whether the same object file has been loaded. This service will erroneously return a `not loaded` condition if the path-name to the object file is expressed differently than it was on a previous load request.

If this flag is set, no object file is loaded and the **LD_SINGLELOAD** and **LD_KERNELEX** flags are ignored, if set.

libpathp
Points to a character string containing the search path to use for finding object files required to complete symbol resolution for this load. If the parameter is null, the search path is set from the specification in the object file header for the object file specified by the *pathp* parameter.

kmidp
Points to an area where the kernel module ID associated with this load of the specified module is to be returned. The data in this area is not valid if the **kmod_load** service returns a nonzero return code.

Description

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

You can specify flags to request a single load, which ensures that only one copy of the object file is loaded into the kernel. An additional option is simply to query for a given object file (specified by path-name). This allows the user to determine if a module is already loaded and then access its assigned kernel module ID.

The **kmod_load** service also provides load-time symbol resolution of the loaded module's imported symbols. The **kmod_load** service loads additional kernel object modules if required for symbol resolution.

Loader Symbol Binding Support

Symbols imported from the kernel name space are resolved with symbols that exist in the kernel name space at the time of the load. (Symbols are imported from the kernel name space by specifying the **#!/unix** character string as the first field in an import list at link-edit time.)

Kernel modules can also import symbols from other kernel object modules. These other kernel object modules are loaded along with the specified object module if they are needed to resolve the imported symbols.

Any symbols exported by the specified kernel object module are added to the kernel name space if the *flags* parameter has the **LD_KERNELEX** flag set. This makes the symbols available to other subsequently loaded kernel object modules. Kernel object modules loaded on behalf of the specified kernel object module (to resolve imported symbols) do not have their exported symbols added to the kernel name space.

Kernel export symbols specified (at link-edit time) with the **SYSCALL** keyword in the primary module's export list are added to the system call table. These kernel export symbols are available to application programs as system calls.

Finding Shared Object Modules for Resolving Symbol References

The search path search string is taken from the module header of the object module specified by the *pathp* parameter if the *libpathp* parameter is null. The module header of the object module specified by the *pathp* parameter is used.

If the module header contains an unqualified base file name for the symbol (no / [slash] characters in the name), a search string is used to find the location of the shared object module required to resolve the import. This search string can be taken from one of two places. If the *libpathp* parameter on the call to the **kmod_load** service is not null, then it points to a character string specifying the search path to be used. However, if the *libpathp* parameter is null, then the search path is to be taken from the module header for the object module specified by the *pathp* parameter.

The search path specification found in object modules loaded to resolve imported symbols is not used. The kernel loader service does not support deferred symbol resolution. The load of the kernel module is terminated with an error if any imported symbols cannot be resolved.

Execution Environment

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

Return Values

If the object file is loaded without error, the module ID is returned in the location pointed to by the *kmidp* parameter and the return code is set to 0.

Error Codes

If an error results, the module is not loaded, and no kernel module ID is returned. The return code is set to one of the following return values:

Return Value	Description
EACCES	Indicates that an object module to be loaded is not an ordinary file or that the mode of the object module file denies read-only access.
EACCES	Search permission is denied on a component of the path prefix.
EFAULT	Indicates that the calling process does not have sufficient authority to access the data area described by the *pathp* or *libpathp* parameters when the **LD_USRPATH** flag is set. This error code is also returned if an I/O error occurs when accessing data in this area.
ENOEXEC	Indicates that the program file has the appropriate access permission, but has an XCOFF indicator that is not valid in its header. The **kmod_load** kernel service supports loading of XCOFF (Extended Common Object File Format) object files only. This error code is also returned if the loader is unable to resolve an imported symbol.
EINVAL	Indicates that the program file has a valid XCOFF indicator in its header, but the header is either damaged or incorrect for the machine on which the file is to be loaded.
ENOMEM	Indicates that the load requires more kernel memory than allowed by the system-imposed maximum.
ETXTBSY	Indicates that the object file is currently open for writing by some process.
ENOTDIR	Indicates that a component of the path prefix is not a directory.
ENOENT	Indicates that no such file or directory exists or the path-name is null.
ESTALE	Indicates that the caller's root or current directory is located in a virtual file system that has been unmounted.
ELOOP	Indicates that too many symbolic links were encountered in translating the *path* or *libpathp* parameter.
ENAMETOOLONG	Indicates that a component of a path-name exceeded 255 characters, or an entire path-name exceeded 1023 characters.
EIO	Indicates that an I/O error occurred during the operation.

Related reference:

"kmod_unload Kernel Service"

Related information:

kmod_util subroutine

Kernel Extension and Device Driver Management Kernel Services

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

Item	Description
kmid	Specifies the kernel module ID of the object file to be unloaded. This kernel module ID is returned when using the **kmod_load** kernel service.
flags	Flags specifying unload options. The following flag is defined:

0 Unloads the object module specified by its *kmid* parameter and any object modules that were loaded as a result of loading the specified object file if this file is not still in use.

Description

The **kmod_unload** kernel service unloads a previously loaded kernel extension object file. The object to be unloaded is specified by the *kmid* parameter. Upon successful completion, the following objects are unloaded or marked *unload pending*:

- The specified object file
- Any imported kernel object modules that were loaded as a result of the loading of the specified module

Users of these exports or system calls are modules bound to this module's exported symbols. If there are no users of any of the module's kernel exports or system calls, the module is immediately unloaded. If there are users of this module, the module is not unloaded but marked *unload pending*.

Marking a module *unload pending* removes the module's exported symbols from the kernel name space. Any system calls exported by this module are also removed. This prohibits new users of these symbols. The module is unloaded only when all current users have been unloaded.

If the unload is successfully completed or marked *pending*, a value of 0 is returned. When an error occurs, the specified module and any imported modules are not unloaded. A nonzero return value indicates the error.

Execution Environment

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

Return Values

Item	Description
0	Indicates successful completion.
EINVAL	Indicates that the *kmid* parameter, which specifies the kernel module, is not valid or does not correspond to a currently loaded module.
EBUSY	The *kmid* parameter specifies a kernel extension that is still intercepting system calls.

Related reference:

"kmod_load Kernel Service" on page 279

Related information:

kmod_util subroutine

Kernel Extension and Device Driver Management Kernel Services

kmod_util Kernel Service

Purpose

Registers routines to be called before and after specified system calls are invoked.

Syntax

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

```
int kmod_util ( flags, buffer, blen)
int flags;
void * buffer;
long blen;
```

Parameters

Item	Description
flags	Specifies the operation. Valid values are **KU_INTERCEPT**, **KU_INTERCEPT_STOP**, and **KU_INTERCEPT_CANCEL**.
buffer	Points to a buffer containing a system call interception header and an array of system call interception structures.
blen	Specifies the length of the buffer.

Description

The **kmod_util** kernel service allows system calls to be intercepted. Routines called **pre-sc** functions are specified to be called before the intercepted system call. Routines called **post-sc** functions are specified to be called after the intercepted system call. In addition, a **pre-sc** function is allowed to abort the system call, providing its own return value and preventing subsequent **pre-sc** functions and the system call itself from being called. Similarly, each **post-sc** function may examine and alter the return value. If a system call does not return (e.g., **thread_terminate**), **post-sc** functions are not called.

For each intercepted system call, either a **pre-sc** function, a **post-sc** function, or both, must be specified. If a **pre-sc** function and **post-sc** function are registered for the same system call in the same **kmod_util** invocation, they are considered paired. All **pre-sc** and **post-sc** functions specified in a **kmod_util** call must be defined in the same kernel extension as the caller of the **kmod_util** kernel service. Other kernel extensions, however, can intercept the same system calls. The most recently registered **pre-sc** function is called first, and its paired **post-sc** function is called last.

The interception of a system call is implemented so that all calls to the system call are intercepted, even for existing processes.

It may be necessary to prevent the interception of certain system calls to avoid destabilizing the system. A future version or release of the **kmod_util** kernel service may prevent the interception of additional system calls, and such a change will not be considered a violation of binary compatibility.

The prototype of a **pre-sc** function is

```
int pre_sc(uintptr_t *rc, void *parms, uintptr_t cookie, void *buffer);
```

where *parms* is a pointer to the parameters of the system call, *cookie* is an opaque value specified by the caller of **kmod_util**, *buffer* is a scratch 128-byte buffer for use by the **pre-sc** function and its paired **post-sc** function.

If the **pre-sc** function returns non-zero, the system call is aborted. The *rc* parameter is the address where an alternate return value can be specified. Subsequent **pre_sc** functions are not called, nor is the system call. For **pre-sc** functions already called, their paired **post-sc** functions are called.

The prototype of a post-sc function is

```
void post_sc(uintptr_t *rc, void *parms, uintptr_t cookie,
   void *buffer);
```

The parameters of the **post-sc** function are the same as those of the **pre-sc** function. In particular, the *buffer* parameter is the same buffer that was passed to the paired **pre-sc** function. The return value can be modified by a **post-sc** function.

For calls to the **kmod_util** kernel service, the buffer contains a header and an array of elements about system calls to be intercepted. The layout of these structures is defined in **<sys/sysconfig.h>**.

An array element is ignored if the **KU_IGNORE** flag is set in the **kue_iflag** field. Otherwise, each array element in the input buffer is validated, and if any errors are found, the entire call fails without any partial execution.

Intercepting System Calls

Calls to **kmod_util()** with the **KU_INTERCEPT** flag initiate system call interception.

Stopping System Call Interception

Calls to **kmod_util()** with the **KU_INTERCEPT_STOP** flag suspend the interception of the specified system calls. If a **pre-sc** function has already been called for a specified system call, its paired **post-sc** function will still be called, but future calls to the system call will not invoke either the **pre-sc** or **post-sc** function. It is not valid to stop interception of a system call that was not originally intercepted by the calling kernel extension.

If the interception of a system call has been suspended, it may be resumed by calling the **kmod_util()** function with the **KU_INTERCEPT** flag, as long as the same values are specified, such as the **pre-sc** and **post-sc** functions.

Cancelling System Call Interception

System call interception can be cancelled by specifying the KU_INTERCEPT_CANCEL flag. When interception is cancelled, the **post-sc** function is not called even if its paired **pre-sc** function was called. It is not valid to cancel interception of a system call that was not originally intercepted by the calling kernel extension, but interception can be cancelled without first stopping interception.

Once interception of a system call has been cancelled, it can be intercepted anew by calling the **kmod_util()** function with the **KU_INTERCEPT** flag. Different **pre-sc** and **pre-sc** functions can be specified in this case.

Return Values

If the specified operations can be enacted for all specified system calls, 0 is returned. Otherwise, a non-zero value is returned and no change in the state of system call interception occurs. If an error occurs because of a validation error in a particular array element, the **kue_oflags** field usually identifies the error in more detail.

Error Codes

If an error results, one of the following error values is returned:

Return Value	Description
EINVAL	The flags parameter is not **KU_INTERCEPT, KU_INTERCEPT_STOP**, nor **KU_INTERCEPT_CANCEL**. The fields in the header are invalid or the *blen* parameter is not consistent with the number of array elements. The buffer was invalid. For **KU_INTERCEPT**, at least one of the **pre-sc** and **post-sc** must be supplied for each system call to be intercepted. All **pre-sc** and **post-sc** functions must be in the same kernel extension as the caller of **kmod_util()**.
EBUSY	A request was made to intercept a system call that was already being intercepted.
ENOENT	A request was made to stop or cancel interception of a system call that was not being intercepted.
ENOMEM	Memory could not be allocated to satisfy the request.
ENOTSUPP	One of the specified system calls is not allowed to be intercepted.

Related reference:

"kmod_load Kernel Service" on page 279

"kmod_unload Kernel Service" on page 282

Related information:

Kernel Extension and Device Driver Management Kernel Services

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

Item	Description
msqid	Specifies the message queue ID, which indicates the message queue for which the control operation is being requested for.
cmd	Specifies which control operation is being requested. There are three valid commands.
buf	Points to the **msqid_ds** structure provided by the caller of the **kmsgctl** service. Data is obtained either from this structure or from status returned in this structure, depending on the *cmd* parameter. The **msqid_ds** structure is defined in the **/usr/include/sys/msg.h** file.

Description

The **kmsgctl** kernel service provides a variety of message-queue control operations as specified by the *cmd* parameter. The **kmsgctl** kernel service provides the same functions for user-mode processes in kernel mode as the **msgctl** subroutine performs for kernel processes or user-mode processes in user mode. The **kmsgctl** service can be called by a user-mode process in kernel mode or by a kernel process. A kernel process can also call the **msgctl** subroutine to provide the same function.

The following three commands can be specified with the *cmd* parameter:

Item	Description
IPC_STAT	Sets only documented fields. See the **msgctl** subroutine.
IPC_SET	Sets the value of the following fields of the data structure associated with the *msqid* parameter to the corresponding values found in the structure pointed to by the *buf* parameter: • `msg_perm.uid` • `msg_perm.gid` • `msg_perm.mode` (only the low-order 9 bits) • `msg_qbytes` To perform the **IPC_SET** operation, the current process must have an effective user ID equal to the value of the `msg_perm.uid` or `msg_perm.cuid` field in the data structure associated with the *msqid* parameter. To raise the value of the `msg_qbytes` field, the calling process must have the appropriate system privilege.
IPC_RMID	Removes from the system the message-queue identifier specified by the *msqid* parameter. This operation also destroys both the message queue and the data structure associated with it. To perform this operation, the current process must have an effective user ID equal to the value of the `msg_perm.uid` or `msg_perm.cuid` field in the data structure associated with the *msqid* parameter.

Execution Environment

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

Return Values

Item	Description
0	Indicates successful completion.
EINVAL	Indicates either • The identifier specified by the *msqid* parameter is not a valid message queue identifier. • The command specified by the *cmd* parameter is not a valid command.
EACCES	The command specified by the *cmd* parameter is equal to **IPC_STAT** and read permission is denied to the calling process.
EPERM	The command specified by the *cmd* parameter is equal to **IPC_RMID**, **IPC_SET**, and the effective user ID of the calling process is not equal to that of the value of the `msg_perm.uid` field in the data structure associated with the *msqid* parameter.
EPERM	Indicates the following conditions: • The command specified by the *cmd* parameter is equal to **IPC_SET**. • An attempt is being made to increase to the value of the `msg_qbytes` field, but the calling process does not have the appropriate system privilege.

Related information:

msgctl subroutine

Message Queue Kernel Services

Understanding System Call Execution

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

Item	Description
key	Specifies either a value of **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or a similar algorithm).

Item	Description
msgflg	Specifies that the *msgflg* parameter is constructed by logically ORing one or more of these values: **IPC_CREAT** Creates the data structure if it does not already exist. **IPC_EXCL** Causes the **kmsgget** kernel service to fail if **IPC_CREAT** is also set and the data structure already exists. **S_IRUSR** Permits the process that owns the data structure to read it. **S_IWUSR** Permits the process that owns the data structure to modify it. **S_IRGRP** Permits the process group associated with the data structure to read it. **S_IWGRP** Permits the process group associated with the data structure to modify it. **S_IROTH** Permits others to read the data structure. **S_IWOTH** Permits others to modify the data structure. The values that begin with **S_I**... are defined in the **/usr/include/sys/stat.h** file. They are a subset of the access permissions that apply to files.
msqid	A reference parameter where a valid message-queue ID is returned if the **kmsgget** kernel service is successful.

Description

The **kmsgget** kernel service returns the message-queue identifier specified by the *msqid* parameter associated with the specified *key* parameter value. The **kmsgget** kernel service provides the same functions for user-mode processes in kernel mode as the **msgget** subroutine performs for kernel processes or user-mode processes in user mode. The **kmsgget** service can be called by a user-mode process in kernel mode or by a kernel process. A kernel process can also call the **msgget** subroutine to provide the same function.

Execution Environment

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

Return Values

Item	Description
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:

Item	Description
EACCES	Indicates that a message queue ID exists for the *key* parameter but operation permission as specified by the *msgflg* parameter cannot be granted.
ENOENT	Indicates that a message queue ID does not exist for the *key* parameter and the **IPC_CREAT** command is not set.
ENOSPC	Indicates that a message queue ID is to be created but the system-imposed limit on the maximum number of allowed message queue IDs systemwide will be exceeded.
EEXIST	Indicates that a message queue ID exists for the value specified by the *key* parameter, and both the **IPC_CREAT** and **IPC_EXCL** commands are set.

Related information:

msgget subroutine

Message Queue Kernel Services

Understanding System Call Execution

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;
ssize_t * bytes;
```

Parameters

Item **Description**

msqid
Specifies the message queue from which to read.

msgp
Points to either an **msgxbuf** or an **msgbuf** structure where the message text is placed. The type of structure pointed to is determined by the values of the *flags* parameter. These structures are defined in the **/usr/include/sys/msg.h** file.

msgsz
Specifies the maximum number of bytes of text to be received from the message queue. The received message is truncated to the size specified by the *msgsz* parameter if the message is longer than this size and **MSG_NOERROR** is set in the *msgflg* parameter. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

msgtyp
Specifies the type of message requested as follows:

- If the *msgtyp* parameter is equal to 0, the first message on the queue is received.
- If the *msgtyp* parameter is greater than 0, the first message of the type specified by the *msgtyp* parameter is received.
- If the *msgtyp* parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *msgtyp* parameter is received.

msgflg
Specifies a value of 0, or is constructed by logically ORing one of several values:

MSG_NOERROR
Truncates the message if it is longer than the number of bytes specified by the *msgsz* parameter.

IPC_NOWAIT
Specifies the action to take if a message of the desired type is not on the queue:

- If **IPC_NOWAIT** is set, then the **kmsgrcv** service returns an **ENOMSG** value.
- If **IPC_NOWAIT** is not set, then the calling process suspends execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue ID specified by the *msqid* parameter is removed from the system. When this occurs, the **kmsgrcv** service returns an **EIDRM** value.
  - The calling process receives a signal that is to be caught. In this case, a message is not received and the **kmsgrcv** service returns an **EINTR** value.

Item	Description
flags	Specifies a value of 0 if a normal message receive is to be performed. If an extended message receive is to be performed, this flag should be set to an **XMSG** value. With this flag set, the **kmsgrcv** service functions as the **msgxrcv** subroutine would. Otherwise, the **kmsgrcv** service functions as the **msgrcv** subroutine would.
bytes	Specifies a reference parameter. This parameter contains the number of message-text bytes read from the message queue upon return from the **kmsgrcv** service. If the message is longer than the number of bytes specified by the *msgsz* parameter bytes but **MSG_NOERROR** is not set, then the **kmsgrcv** kernel service fails and returns an **E2BIG** return value.

Description

The **kmsgrcv** kernel service reads a message from the queue specified by the *msqid* parameter and stores the message into the structure pointed to by the *msgp* parameter. The **kmsgrcv** kernel service provides the same functions for user-mode processes in kernel mode as the **msgrcv** and **msgxrcv** subroutines perform for kernel processes or user-mode processes in user mode.

The **kmsgrcv** service can be called by a user-mode process in kernel mode or by a kernel process. A kernel process can also call the **msgrcv** and **msgxrcv** subroutines to provide the same functions.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates that the ID specified by the *msqid* parameter is not a valid message queue ID.
EACCES	Indicates that operation permission is denied to the calling process.
EINVAL	Indicates that the value of the *msgsz* parameter is less than 0.
E2BIG	Indicates that the message text is greater than the maximum length specified by the *msgsz* parameter and **MSG_NOERROR** is not set.
ENOMSG	Indicates that the queue does not contain a message of the desired type and **IPC_NOWAIT** is set.
EINTR	Indicates that the **kmsgrcv** service received a signal.
EIDRM	Indicates that the message queue ID specified by the *msqid* parameter has been removed from the system.

Related information:

msgrcv subroutine

msgxrcv subroutine

Message Queue Kernel Services

kmsgsnd Kernel Service

Purpose

Sends a message using a previously defined message queue.

Syntax

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

int kmsgsnd (msqid, msgp, msgsz, msgflg)
int  msqid;
struct msgbuf *  msgp;
int  msgsz,  msgflg;
```

Parameters

Item	Description
msqid	Specifies the message queue ID that indicates which message queue the message is to be sent on.
msgp	Points to an **msgbuf** structure containing the message. The **msgbuf** structure is defined in the **/usr/include/sys/msg.h** file.
msgsz	Specifies the size of the message to be sent in bytes. The *msgsz* parameter can range from 0 to a system-imposed maximum.
msgflg	Specifies the action to be taken if the message cannot be sent for one of several reasons.

Description

The **kmsgsnd** kernel service sends a message to the queue specified by the *msqid* parameter. The **kmsgsnd** kernel service provides the same functions for user-mode processes in kernel mode as the **msgsnd** subroutine performs for kernel processes or user-mode processes in user mode. The **kmsgsnd** service can be called by a user-mode process in kernel mode or by a kernel process. A kernel process can also call the **msgsnd** subroutine to provide the same function.

There are two reasons why the **kmsgsnd** kernel service cannot send the message:

- The number of bytes already on the queue is equal to the **msg_qbytes** member.
- The total number of messages on all queues systemwide is equal to a system-imposed limit.

There are several actions to take when the **kmsgsnd** kernel service cannot send the message:

- If the *msgflg* parameter is set to **IPC_NOWAIT**, then the message is not sent, and the **kmsgsnd** service fails and returns an **EAGAIN** value.
- If the *msgflg* parameter is 0, then the calling process suspends execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The message queue ID specified by the *msqid* parameter is removed from the system. When this occurs, the **kmsgsnd** service fails and an **EIDRM** value is returned.
  - The calling process receives a signal that is to be caught. In this case, the message is not sent and the calling process resumes execution as described in the **sigaction** kernel service.

Execution Environment

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

The calling process must have write permission to perform the **kmsgsnd** operation.

Return Values

Item	Description
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.

Related information:

msgsnd subroutine

Message Queue Kernel Services

Understanding System Call Execution

kra_attachrset Subroutine

Purpose

Attaches a work component to a resource set.

Syntax

```
#include <sys/rset.h>
int kra_attachrset (rstype, rsid, rset, flags)
rstype_t rstype;
rsid_t rsid;
rsethandle_t rset;
unsigned int flags;
```

Description

The **kra_attachrset** subroutine attaches a work component specified by the *rstype* and *rsid* parameters to a resource set specified by the *rset* parameter.

The work component is an existing process identified by the process ID or an existing kernel thread identified by the kernel thread ID (tid). A process ID or thread ID value of RS_MYSELF indicates the attachment applies to the current process or the current kernel thread, respectively.

The following conditions must be met to successfully attach a process to a resource set:

- The resource set must contain processors that are available in the system.
- The calling process must either have root authority or have CAP_NUMA_ATTACH capability.
- The calling process must either have root authority or the same effective userid as the target process.
- The target process must not contain any threads that have bindprocessor bindings to a processor.
- The resource set must be contained in (be a subset of) the target process' partition resource set.
- The resource set must be a superset of all the thread's *rset* in the target process.

The following conditions must be met to successfully attach a kernel thread to a resource set:

- The resource set must contain processors that are available in the system.
- The calling process must either have root authority or have CAP_NUMA_ATTACH capability.
- The calling process must either have root authority or the same effective userid as the target process.
- The target thread must not have bindprocessor bindings to a processor.
- The resource set must be contained in (be a subset of) the target thread's process effective and partition resource set.

If any of these conditions are not met, the attachment will fail.

Once a process is attached to a resource set, the threads in the process will only run on processors contained in the resource set. Once a kernel thread is attached to a resource set, that thread will only run on processors contained in the resource set.

The *flags* parameter can be set to indicate the policy for using the resources contained in the resource set specified in the *rset* parameter. The only supported scheduling policy is R_ATTACH_STRSET, which is useful only when the processors of the system are running in simultaneous multithreading mode. Processors like the POWER5 support simultaneous multithreading, where each physical processor has two execution engines, called *hardware threads*. Each hardware thread is essentially equivalent to a single CPU, and each is identified as a separate CPU in a resource set. The R_ATTACH_STRSET flag indicates that the process is to be scheduled with a single-threaded policy; namely, that it should be scheduled on only one hardware thread per physical processor. If this flag is specified, then all of the available processors indicated in the resource set must be of exclusive use. A new resource set, called an *ST resource set*, is constructed from the specified resource set and attached to the process according to the following rules:

- All offline processors are ignored.
- If all the hardware threads (CPUs) of a physical processor (when running in simultaneous multithreading mode, there will be more than one active hardware thread per physical processor) are not included in the specified resource set, the other CPUs of the processor are ignored when constructing the ST resource set.
- Only one CPU (hardware thread) resource per physical processor is included in the ST resource set.

Parameters

Item **Description**

rstype
Specifies the type of work component to be attached to the resource set specified by the *rset* parameter. The *rstype* parameter must be the following value, defined in **rset.h**:

- R_PROCESS: existing process
- R_THREAD: existing kernel thread

rsid
Identifies the work component to be attached to the resource set specified by the *rset* parameter. The *rsid* parameter must be the following:

- Process ID (for *rstype* of R_PROCESS): set the *rsid_t at_pid* field to the desired process' process ID.
- Kernel thread ID (for *rstype* of R_THREAD): set the *rsid_t at_tid* field to the desired kernel thread's thread ID.

rset
Specifies which work component (specified by the *rstype* and *rsid* parameters) to attach to the resource set.

flags

Specifies the scheduling policy for the work component being attached.

The only supported value is R_ATTACH_STRSET value, which is only applicable if the *rstype* parameter is set to R_PROCESS. The R_ATTACH_STRSET value indicates that the process is to be scheduled with a single-threaded policy (only on one hardware thread per physical processor).

Return Values

Upon successful completion, the **kra_attachrset** subroutine returns a 0. If unsuccessful, one or more of the following are true:

Item **Description**

EINVAL
One of the following is true:

- The *flags* parameter contains an invalid value.
- The *rstype* parameter contains an invalid type qualifier.
- The R_ATTACH_STRSET *flags* parameter is specified and one or more processors in the *rset* parameter are not assigned for exclusive use.

ENODEV
The resource set specified by the *rset* parameter does not contain any available processors, or the R_ATTACH_STRSET *flags* parameter is specified and the constructed ST resource set does not have any available processors.

ESRCH
The process or kernel thread identified by the *rstype* and *rsid* parameters does not exist.

EPERM
One of the following is true:

- If the *rstype* is R_PROCESS, either the resource set specified by the *rset* parameter is not included in the partition resource set of the process identified by the *rstype* and *rsid* parameters, or any of the thread's R_THREAD *rset* in this process is not a subset of the resource set specified by the *rset* parameter.
- If the *rstype* is R_THREAD, the resource set specified by the *rset* parameter is not included in the target thread's process effective or partition (real) resource set.
- The calling process has neither root authority nor CAP_NUMA_ATTACH attachment privilege.
- The calling process has neither root authority nor the same effective user ID as the process identified by the *rstype* and *rsid* parameters.
- The process or thread identified by the *rstype* and *rsid* parameters has one or more threads with a bindprocessor processor binding.

Related reference:

“kra_getrset Subroutine” on page 296

“kra_detachrset Subroutine” on page 295

Related information:

Exclusive use processor resource sets

kra_creatp Subroutine

Purpose

Creates a new kernel process and attaches it to a resource set.

Syntax

```
#include <sys/rset.h>
int kra_creatp (pid, rstype, rsid, flags)
pid_t *pid;
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
```

Description

The **kra_creatp** kernel service creates a new kernel process and attaches it to a resource set. The **kra_creatp** kernel service attaches the new kernel process to the resource set specified by the *rstype* and *rsid* parameters.

The **kra_creatp** kernel service is similar to the **creatp** kernel service.

The following conditions must be met to successfully attach a kernel process to a resource set:

- The resource set must contain processors that are available in the system.
- The calling process must either have root authority or have CAP_NUMA_ATTACH capability.
- The calling thread must not have a bindprocessor binding to a processor.
- The resource set must be contained in the calling process' partition resource set.

Note: When the **creatp** kernel service is used, the new kernel process inherits its parent's resource set attachments.

Parameters

Item	Description
pid	Pointer to a **pid_t** field to receive the process ID of the new kernel process.
rstype	Specifies the type of resource the new process will be attached to. This parameter must be the following value, defined in **rset.h**. • R_RSET: resource set.
rsid	Identifies the resource set the new process will be attached to. • Resource set ID (for *rstype* of R_RSET): set the *rsid_t at_rset* field to the desired resource set.
flags	Reserved for future use. Specify as 0.

Return Values

Upon successful completion, the **kra_creatp** kernel service returns a 0. If unsuccessful, one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *rstype* parameter contains an invalid type identifier. • The *flags* parameter contains an invalid flags value.
ENODEV	The specified resource set does not contain any available processors.
EFAULT	Invalid address.
EPERM	One of the following is true: • The calling process has neither root authority nor CAP_NUMA_ATTACH attachment privilege. • The calling process contains one or more threads with a bindprocessor processor binding. • The specified resource set is not included in the calling process' partition resource set.
ENOMEM	Memory not available.

Related reference:

"creatp Kernel Service" on page 53
"initp Kernel Service" on page 220
"kra_attachrset Subroutine" on page 292

kra_detachrset Subroutine

Purpose

Detaches a work component from a resource set.

Syntax

```
#include <sys/rset.h>
int kra_detachrset (rstype, rsid, flags)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
```

Description

The **kra_detachrset** subroutine detaches a work component specified by *rstype* and *rsid* from a resource set.

The work component is an existing process identified by the process ID or an existing kernel thread identified by the kernel thread ID (tid). A process ID or thread ID value of RS_MYSELF indicates the detach command applies to the current process or the current kernel thread, respectively.

The following conditions must be met to detach a process or kernel thread from a resource set:

- The calling process must either have root authority or have CAP_NUMA_ATTACH capability.
- The calling process must either have root authority or the same effective userid as the target process.

If these conditions are not met, the operation will fail.

Once a process is detached from a resource set, the threads in the process can run on all available processors contained in the process' partition resource set. Once a kernel thread is detached from a resource set, that thread can run on all available processors contained in its process effective or partition resource set.

Parameters

Item **Description**

rstype Specifies the type of work component to be detached from to the resource set specified by *rset*. This parameter must be the following value, defined in **rset.h**:

- R_PROCESS: existing process
- R_THREAD: existing kernel thread

rsid Identifies the work component to be attached to the resource set specified by *rset*. This parameter must be the following:

- Process ID (for *rstype* of R_PROCESS): set the *rsid_t at_pid* field to the desired process' process ID.
- Kernel thread ID (for *rstype* of R_THREAD): set the *rsid_t at_tid* field to the desired kernel thread's thread ID.

flags For *rstype* of R_PROCESS, the R_DETACH_ALLTHRDS indicates that R_THREAD *rsets* are detached from all threads in a specified process. The process' effective *rset* is not detached in this case. Reserved for future use. Specify as `0`.

Return Values

Upon successful completion, the **kra_detachrset** subroutine returns a 0. If unsuccessful, one or more of the following are true:

Item **Description**

EINVAL One of the following is true:

- The *flags* parameter contains an invalid value.
- The *rstype* contains an invalid type qualifier.

ESRCH The process or kernel thread identified by the *rstype* and *rsid* parameters does not exist.

EPERM One of the following is true:

- The calling process has neither root authority nor CAP_NUMA_ATTACH attachment privilege.
- The calling process has neither root authority nor the same effective user ID as the process identified by the *rstype* and *rsid* parameters.

Related reference:

"kra_attachrset Subroutine" on page 292

kra_getrset Subroutine

Purpose

Gets the resource set to which a work component is attached.

Syntax

```
#include <sys/rset.h>
int kra_getrset (rstype, rsid, flags, rset, rset_type)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
rsethandle_t rset;
unsigned int *rset_type;
```

Description

The **kra_getrset** subroutine returns the resource set to which a specified work component is attached.

The work component is an existing process identified by the process ID or an existing kernel thread identified by the kernel thread ID (tid). A process ID or thread ID value of RS_MYSELF indicates the resource set attached to the current process or the current kernel thread, respectively, is requested.

Upon successful completion, one of the following types of resource set is returned into the *rset_type* parameter:

- A value of RS_EFFECTIVE_RSET indicates the process was explicitly attached to the resource set. This may have been done with the **kra_attachrset** subroutine.

- A value of RS_PARTITION_RSET indicates the process was not explicitly attached to a resource set. However, the process had an explicitly set partition resource set. This may be set with the **krs_setpartition** subroutine or through the use of WLM work classes with resource sets.
- A value of RS_DEFAULT_RSET indicates the process was not explicitly attached to a resource set nor did it have an explicitly set partition resource set. The system default resource set is returned.
- A value of RS_THREAD_RSET indicates the kernel thread was explicitly attached to the resource set. This might have been done with the **ra_attachrset** subroutine.

Parameters

Item	Description
rstype	Specifies the type of the work component whose resource set attachment is requested. This parameter must be the following value, defined in **rset.h**: • R_PROCESS: existing process • R_THREAD: existing kernel thread
rsid	Identifies the work component whose resource set attachment is requested. This parameter must be the following: • Process ID (for *rstype* of R_PROCESS): set the *rsid_t at_pid* field to the desired process' process ID. • Kernel thread ID (for *rstype* of R_THREAD): set the *rsid_t at_tid* field to the desired kernel thread's thread ID.
flags	Reserved for future use. Specify as 0.
rset	Specifies the resource set to receive the work component's resource set.
rset_type	Points to an unsigned integer field to receive the resource set type.

Return Values

Upon successful completion, the **kra_getrset** subroutine returns a 0. If unsuccessful, one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *flags* parameter contains an invalid value. • The *rstype* parameter contains an invalid type qualifier.
EFAULT	Invalid address.
ESRCH	The process or kernel thread identified by the *rstype* and *rsid* parameters does not exist.

Related reference:

"krs_getpartition Subroutine" on page 301

krs_alloc Subroutine

Purpose

Allocates a resource set and returns its handle.

Syntax

```
#include <sys/rset.h>
int krs_alloc (rset, flags)
rsethandle_t *rset;
unsigned int flags;
```

Description

The **krs_alloc** subroutine allocates a resource set and initializes it according to the information specified by the *flags* parameter. The value of the *flags* parameter determines how the new resource set is initialized.

Parameters

Item	Description
rset	Points to an **rsethandle_t** where the resource set handle is stored on successful completion.
flags	Specifies how the new resource set is initialized. It takes one of the following values, defined in **rset.h**: • **RS_EMPTY** (or 0 value): The resource set is initialized to contain no resources. • **RS_SYSTEM**: The resource set is initialized to contain available system resources. • **RS_ALL**: The resource set is initialized to contain all resources. • **RS_PARTITION**: The resource set is initialized to contain the resources in the caller's process partition resource set.

Return Values

Upon successful completion, the **krs_alloc** subroutine returns a 0. If unsuccessful, one or more of the following is returned:

Item	Description
EINVAL	The *flags* parameter contains an invalid value.
ENOMEM	There is not enough space to create the data structures related to the resource set.

Related reference:

"krs_free Subroutine"

"krs_getinfo Subroutine" on page 300

"krs_init Subroutine" on page 303

krs_free Subroutine

Purpose

Frees a resource set.

Syntax

```
#include <sys/rset.h>
void krs_free(rset)
rsethandle_t rset;
```

Description

The **krs_free** subroutine frees a resource set identified by the *rset* parameter. The resource set must have been allocated by the **krs_alloc** subroutine.

Parameters

Item	Description
rset	Specifies the resource set whose memory will be freed.

Related reference:

"krs_alloc Subroutine" on page 297

krs_getassociativity Subroutine

Purpose

Gets the hardware associativity values for a resource.

Syntax

```
#include <sys/rset.h>
int krs_getassociativity (type, id, assoc_array, array_size)
unsigned int type;
unsigned int id;
unsigned int *assoc_array;
unsigned int array_size;
```

Description

The **krs_getassociativity** subroutine returns the array of hardware associativity values for a specified resource.

This is a special purpose subroutine intended for specialized root applications needing the hardware associativity value information. The **krs_getinfo**, **krs_getrad**, and **krs_numrads** subroutines are provided for typical applications to discover system hardware topology.

The calling process must have root authority to get hardware associativity values.

Parameters

Item	Description
type	Specifies the resource type whose associativity values are requested. The only value supported to retrieve values for a processor is R_PROCS.
id	Specifies the logical resource id whose associativity values are requested.
assoc_array	Specifies the address of an array of unsigned integers to receive the associativity values.
array_size	Specifies the number of unsigned integers in *assoc_array*.

Return Values

Upon successful completion, the **krs_getassociativity** subroutine returns a 0. The *assoc_array* parameter array contains the resource's associativity values. The first entry in the array indicates the number of associativity values returned. If the hardware system does not provide system topology data, a value of 0 is returned in the first array entry. If unsuccessful, one or more of the following are returned:

Item	Description
EINVAL	One of the following occurred: • The *array_size* parameter was specified as 0. • An invalid *type* parameter was specified.
ENODEV	The resource specified by the *id* parameter does not exist.
EFAULT	Invalid address.
EPERM	The calling process does not have root authority.

Related reference:

"krs_getinfo Subroutine" on page 300

"krs_getrad Subroutine" on page 302

"krs_numrads Subroutine" on page 304

krs_get_homesrad Subroutine

Purpose

Gets the currently running thread's home SRADID (Scheduler Resource Allocation Domain Identifier).

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
sradid_t krs_get_homesrad(void)
```

Description

The krs_get_homesrad is a kernel service and if the ENHANCED_AFFINITY services are enabled, the **krs_get_homesrad** subroutine returns the home SRADID of the currently running thread. If the ENHANCED_AFFINITY services are not enabled, the **krs_get_homesrad** subroutine returns SRADID_ANY. SRADID is the index of a RAD (Resource Allocation Domain) at the R_SRADSDL system detail level.

Return Values

If the ENHANCED_AFFINITY services are enabled, the home SRADID of the currently running thread is returned. Otherwise, SRADID_ANY is returned.

Related reference:

"krs_getrad Subroutine" on page 302

krs_getinfo Subroutine

Purpose

Gets information about a resource set.

Syntax

```
#include <sys/rset.h>
int krs_getinfo(rset, info_type, flags, result)
rsethandle_t rset;
rsinfo_t info_type;
unsigned int flags;
int *result;
```

Description

The **krs_getinfo** subroutine retrieves information about the resource set identified by the *rset* parameter. Depending on the value of the *info_type* parameter, the **krs_getinfo** subroutine returns information about the number of available processors, the number of available memory pools, or the amount of available memory contained in the resource *rset*.

The subroutine can also return global system information such as the maximum system detail level, the symmetric multiprocessor (SMP) and multiple chip module (MCM) system detail levels, and the maximum number of processor or memory pool resources in a resource set.

Parameters

Item	Description
rset	Specifies a resource set handle of a resource set the information should be retrieved from. This parameter is not meaningful if the *info_type* parameter is R_MAXSDL, R_MAXPROCS, R_MAXMEMPS, R_SMPSDL, or R_MCMSDL.

Item **Description**

info_type Specifies the type of information being requested. One of the following values (defined in **rset.h**) can be used:

- **R_NUMPROCS**: The number of available processors in the resource set is returned.
- **R_NUMMEMPS**: The number of available memory pools in the resource set is returned.
- **R_MEMSIZE**: The amount of available memory (in MB) contained in the resource set is returned.
- **R_MAXSDL**: The maximum system detail level of the system is returned.
- **R_MAXPROCS**: The maximum number of processors that may be contained in a resource set is returned.
- **R_MAXMEMPS**: The maximum number of memory pools that may be contained in a resource set is returned.
- **R_SMPSDL**: The system detail level that corresponds to the traditional notion of an SMP is returned. A system detail level of 0 is returned if the hardware system does not provide system topology data.
- **R_MCMSDL**: The system detail level that corresponds to resources packaged in an MCM is returned. A system detail level of 0 is returned if the hardware system does not have MCMs or does not provide system topology data.

flags Reserved for future use. Must be specified as 0.

result Points to an integer where the result is stored on successful completion.

Return Values

Upon successful completion, the **krs_getinfo** subroutine returns a 0, and the *result* field contains the requested information. If unsuccessful, one or more of the following are returned:

Item **Description**

EINVAL One of the following is true:

- The *info_type* parameter specifies an invalid resource type value.
- The *flags* parameter was not specified as 0.

EFAULT Invalid address.

Related reference:

"krs_numrads Subroutine" on page 304

krs_getpartition Subroutine

Purpose

Gets the partition resource set to which a process is attached.

Syntax

```
#include <sys/rset.h>
int krs_getpartition (pid, flags, rset, rset_type)
pid_t pid;
unsigned int flags;
rsethandle_t rset;
unsigned int *rset_type;
```

Description

The **krs_getpartition** subroutine returns the partition resource set attached to the specified process. A process ID value of RS_MYSELF indicates the partition resource set attached to the current process is requested.

Upon successful completion, the type of resource set is returned into the *rset_type* parameter.

A value of RS_PARTITION_RSET indicates the process has a partition resource set that is set explicitly. This may be set with the **krs_setpartition** subroutine or through the use of WLM work classes with resource sets.

A value of RS_DEFAULT_RSET indicates the process did not have an explicitly set partition resource set. The system default resource set is returned.

Parameters

Item	Description
pid	Specifies the process ID whose partition *rset* is requested.
flags	Reserved for future use. Specify as 0.
rset	Specifies the resource set to receive the process' partition resource set.
rset_type	Points to an unsigned integer field to receive the resource set type.

Return Values

Upon successful completion, the **krs_getpartition** subroutine returns a 0. If unsuccessful, one or more of the following are true:

Item	Description
EFAULT	Invalid address.
ESRCH	The process identified by the *pid* parameter does not exist.

Related reference:

"kra_getrset Subroutine" on page 296

krs_getrad Subroutine

Purpose

Returns a system resource allocation domain (RAD) contained in an input resource set.

Syntax

```
#include <sys/rset.h>
int krs_getrad (rad, sdl, index, flags)
rsethandle_t rad;
unsigned int sdl;
unsigned int index;
unsigned int flags;
```

Description

The **krs_getrad** subroutine returns a system RAD at a specified system detail level and index.

The system RAD is specified by system detail level *sdl* and index number *index*.

The *rad* parameter must be allocated (using the **krs_alloc** subroutine) prior to calling the **krs_getrad** subroutine.

Parameters

Item	Description
rad	Specifies a resource set handle to receive the desired system RAD.
sdl	Specifies the system detail level of the desired system RAD.
index	Specifies the index of the system RAD that should be returned from among those at the specified *sdl*. This parameter must belong to the the **[0, krs_numrads(*rset*, *sdl*, *flags*)- 1]** interval.
flags	Reserved for future use. Specify as 0.

Return Values

Upon successful completion, the **krs_getrad** subroutine returns a 0. If unsuccessful, one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *flags* parameter contains an invalid value. • The *sdl* parameter is greater than the maximum system detail level. • The RAD specified by the *index* parameter does not exist at the system detail level specified by the *sdl* parameter.
EFAULT	Invalid address.

Related reference:

"krs_numrads Subroutine" on page 304

"krs_getinfo Subroutine" on page 300

"krs_alloc Subroutine" on page 297

krs_init Subroutine

Purpose

Initializes a previously allocated resource set.

Syntax

```
#include <sys/rset.h>
int krs_init (rset, flags)
rsethandle_t rset;
unsigned int flags;
```

Description

The **krs_init** subroutine initializes a previously allocated resource set. The resource set is initialized according to information specified by the *flags* parameter.

Parameters

Item	Description
rset	Specifies the handle of the resource set to initialize.
flags	Specifies how the resource set is initialized. It takes one of the following values, defined in **rset.h**: • **RS_EMPTY**: The resource set is initialized to contain no resources. • **RS_SYSTEM**: The resource set is initialized to contain available system resources. • **RS_ALL**: The resource set is initialized to contain all resources. • **RS_PARTITION**: The resource set is initialized to contain the resources in the caller's process partition resource set.

Return Values

Upon successful completion, the krs_init subroutine returns a 0. If unsuccessful, the following is returned:

Item	Description
EINVAL	The *flags* parameter contains an invalid value.

Related reference:

"krs_alloc Subroutine" on page 297

krs_numrads Subroutine

Purpose

Returns the number of system resource allocation domains (RADs) that have available resources.

Syntax

```
#include <sys/rset.h>
int krs_numrads(rset, sdl, flags)
rsethandle_t rset;
unsigned int sdl;
unsigned int flags;
```

Description

The **krs_numrads** subroutine returns the number of system RADs at system detail level *sdl*, that have available resources contained in the resource set identified by the *rset* parameter.

The number of atomic RADs contained in the *rset* parameter is returned if the *sdl* parameter is equal to the maximum system detail level.

Parameters

Item	Description
rset	Specifies the resource set handle for the resource set being queried.
sdl	Specifies the system detail level in which the caller is interested.
flags	Reserved for future use. Specify as 0.

Return Values

Upon successful completion, the number of RADs is returned. If unsuccessful, a -1 is returned and one or more of the following are true:

- The *flags* parameter contains an invalid value.
- The *sdl* parameter is greater than the maximum system detail level.

Related reference:

"krs_getrad Subroutine" on page 302

"krs_getinfo Subroutine" on page 300

krs_op Subroutine

Purpose

Performs a set of operations on one or two resource sets.

Syntax

```
#include <sys/rset.h>
int krs_op (command, rset1, rset2, flags, id)
unsigned int command;
rsethandle_t rset1, rset2;
unsigned int flags;
unsigned int id;
```

Description

The **krs_op** subroutine performs the operation specified by the *command* parameter on resource set *rset1*, or both resource sets *rset1* and *rset2*.

Parameters

Item **Description**

command
Specifies the operation to apply to the resource sets identified by *rset1* and *rset2*. One of the following values, defined in **rset.h**, can be used:

- **RS_UNION**: The resources contained in either *rset1* or *rset2* are stored in *rset2*.
- **RS_INTERSECTION**: The resources that are contained in both *rset1* and *rset2* are stored in *rset2*.
- **RS_EXCLUSION**: The resources in *rset1* that are also in *rset2* are removed from *rset2*. On completion, *rset2* contains all the resources that were contained in *rset2* but were not contained in *rset1*.
- **RS_COPY**: All resources in *rset1* whose type is *flags* are stored in *rset2*. If *rset1* contains no resources of this type, *rset2* will be empty. The previous content of *rset2* is lost, while the content of *rset1* is unchanged.
- **RS_ISEMPTY**: Test if resource set *rset1* is empty.
- **RS_ISEQUAL**: Test if resource sets *rset1* and *rset2* are equal.
- **RS_ISCONTAINED**: Test if all resources in resource set *rset1* are also contained in resource set *rset2*.
- **RS_TESTRESOURCE**: Test if the resource whose type is *flags* and index is *id* is contained in resource set *rset1*.
- **RS_ADDRESOURCE**: Add the resource whose type is *flags* and index is *id* to resource set *rset1*.
- **RS_DELRESOURCE**: Delete the resource whose type is *flags* and index is *id* from resource set *rset1*.
- **RS_STSET**: Constructs an ST resource set by including only one hardware thread per physical processor included in *rset1* and stores it in *rset2*. Only available processors are considered when constructing the ST resource set.

rset1
Specifies the resource set handle for the first of the resource sets involved in the *command* operation.

rset2
Specifies the resource set handle for the second of the resource sets involved in the *command* operation. This resource set is also used, on return, to store the result of the operation, and its previous content is lost. The *rset2* parameter is ignored on the RS_ISEMPTY, RS_TESTRESOURCE, RS_ADDRESOURCE, and RS_DELRESOURCE commands.

flags
When combined with the RS_COPY command, the *flags* parameter specifies the type of the resources that will be copied from *rset1* to *rset2*. This parameter is constructed by logically ORing one or more of the following values, defined in **rset.h**:

- **R_PROCS:** processors
- **R_MEMPS:** memory pools
- **R_ALL_RESOURCES:** processors and memory pools

If none of the above are specified for *flags*, R_ALL_RESOURCES is assumed.

id
On the RS_TESTRESOURCE, RS_ADDRESOURCE, and RS_DELRESOURCE commands, the *id* parameter specifies the index of the resource to be tested, added, or deleted. This parameter is ignored on the other commands.

Return Values

Item **Description**

0
Successful completion. The tested condition is not met for the RS_ISEMPTY, RS_ISEQUAL, RS_ISCONTAINED, and RS_TESTRESOURCE commands.

1
Successful completion. The tested condition is met for the RS_ISEMPTY, RS_ISEQUAL, RS_ISCONTAINED, and RS_TESTRESOURCE commands.

-1
Unsuccessful completion. One or more of the following are true:

- *rset1* identifies an invalid resource set.
- *rset2* identifies an invalid resource set.
- *command* identifies an invalid operation.
- *flags* identifies an invalid resource type.
- *id* specifies a resource index that is too large.
- Invalid address.

krs_setpartition Subroutine

Purpose

Sets the partition resource set of a process.

Syntax

```
#include <sys/rset.h>
int krs_setpartition(pid, rset, flags)
pid_t pid;
rsethandle_t rset;
unsigned int flags;
```

Description

The **krs_setpartition** subroutine sets a process' partition resource set. The subroutine can also be used to remove a process' partition resource set.

The partition resource set limits the threads in a process to running only on the processors contained in the partition resource set.

The work component is an existing process identified by process ID. A process ID value of RS_MYSELF indicates the attachment applies to the current process.

The following conditions must be met to set a process' partition resource set:

- The calling process must have root authority.
- The resource set must contain processors that are available in the system.
- The new partition resource set must be equal to, or a superset of the target process' effective resource set.
- The target process must not contain any threads that have bindprocessor bindings to a processor.

The *flags* parameter can be set to indicate the policy for using the resources contained in the resource set specified in the *rset* parameter. The only supported scheduling policy is **R_ATTACH_STRSET**, which is useful only when the processors of the system are running in simultaneous multithreading mode. Processors like the POWER5 support simultaneous multithreading, where each physical processor has two execution engines, called *hardware threads*. Each hardware thread is essentially equivalent to a single CPU, and each is identified as a separate CPU in a resource set. The **R_ATTACH_STRSET** flag indicates that the process is to be scheduled with a single-threaded policy; namely, that it should be scheduled on only one hardware thread per physical processor. If this flag is specified, then all of the available processors indicated in the resource set must be of exclusive use. A new resource set, called an *ST resource set*, is constructed from the specified resource set and attached to the process according to the following rules:

- All offline processors are ignored.
- If all the hardware threads (CPUs) of a physical processor (when running in simultaneous multithreading mode, there will be more than one active hardware thread per physical processor) are not included in the specified resource set, the other CPUs of the processor are ignored when constructing the ST resource set.
- Only one CPU (hardware thread) resource per physical processor is included in the ST resource set.

Parameters

Item **Description**

pid Specifies the process ID of the process whose partition resource set is to be set. A value of RS_MYSELF indicates the current process' partition resource set should be set.

rset Specifies the partition resource set to be set. A value of RS_DEFAULT indicates the process' partition resource set should be removed.

flags Specifies the policy to use for the process. A value of R_ATTACH_STRSET indicates that the process is to be scheduled with a single-threaded policy (only on one hardware thread per physical processor).

Return Values

Upon successful completion, the **krs_setpartition** subroutine returns a 0. If unsuccessful, one or more of the following are true:

Item **Description**

EINVAL The R_ATTACH_STRSET *flags* parameter is specified and one or more processors in the *rset* parameter are not assigned for exclusive use.

ENODEV The resource set specified by the *rset* parameter does not contain any available processors, or the R_ATTACH_STRSET *flags* parameter is specified and the constructed ST resource set does not have any available processors.

ESRCH The process identified by the *pid* parameter does not exist.

EFAULT Invalid address.

ENOMEM Memory not available.

EPERM One of the following is true:

- The calling process does not have root authority.
- The process identified by the *pid* parameter has one or more threads with a bindprocessor processor binding.
- The process identified by the *pid* parameter has an effective resource set and the new partition resource set identified by the *rset* parameter does not contain all of the effective resource set's resources.

Related reference:

"krs_getpartition Subroutine" on page 301

"kra_attachrset Subroutine" on page 292

Related information:

Exclusive use processor resource sets

ksettickd Kernel Service

Purpose

Sets the current status of the systemwide timer-adjustment values.

Syntax

```
#include <sys/types.h>
int ksettickd (timed, tickd, time_adjusted)
int *timed;
int *tickd;
int *time_adjusted;
```

Parameters

Item	Description
timed	Specifies the number of microseconds by which the systemwide timer is to be adjusted unless set to a null pointer.
tickd	Specifies the adjustment rate of the systemwide timer unless set to a null pointer. This rate determines the number of microseconds that the systemwide timer is adjusted with each timer tick. Adjustment continues until the time has been corrected by the amount specified by the *timed* parameter.
time_adjusted	Sets the kernel-maintained time adjusted flag to True or False. If the *time_adjusted* parameter is a null pointer, calling the **ksettickd** kernel service always sets the kernel's *time_adjusted* parameter to False.

Description

The **ksettickd** kernel service provides kernel extensions with the capability to update the *time_adjusted* parameter, and set or change the systemwide time-of-day timer adjustment amount and rate. The timer-adjustment values indicated by the *timed* and *tickd* parameters are the same values used by the **adjtime** subroutine. A call to the **settimer** or **adjtime** subroutine for the systemwide time-of-day timer sets the *time_adjusted* parameter to True, as read by the **kgettickd** kernel service.

This kernel service is typically used only by kernel extensions providing time synchronization functions such as coordinated network time where the **adjtime** subroutine is insufficient.

Note: The **ksettickd** service provides no serialization with respect to the **adjtime** and **settimer** subroutines, the **ksettimer** kernel service, or the timer interrupt handler, all of which also use and update these values. The caller of this kernel service must provide the necessary serialization to ensure appropriate operation.

Execution Environment

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

Return Value

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

Related reference:

"kgettickd Kernel Service" on page 270

Related information:

adjtime subroutine

Using Fine Granularity Timer Services and Structures

ksettimer Kernel Service

Purpose

Sets the systemwide time-of-day timer.

Syntax

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

int ksettimer (nct)
struct timestruc_t *nct;
```

Parameter

Item	Description
nct	Points to a **timestruc_t** structure, which contains the new current time to be set. The nanoseconds member of this structure is valid only if greater than or equal to 0, and less than the number of nanoseconds in a second.

Description

The **ksettimer** kernel service provides a kernel extension with the capability to set the systemwide time-of-day timer. Kernel extensions typically use this kernel service to support network coordinated time, which is the periodic synchronization of all system clocks to a common time by a time server or set of time servers on a network. The newly set "current" time must represent the amount of time since 00:00:00 GMT, January 1, 1970.

Execution Environment

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

Return Values

Item	Description
0	Indicates success.
EINVAL	Indicates that the new current time specified by the *nct* parameter is outside the range of the systemwide timer.
EIO	Indicates that an error occurred while this kernel service was accessing the timer device.

Related information:

Using Fine Granularity Timer Services and Structures

Timer and Time-of-Day Kernel Services

kthread_kill Kernel Service

Purpose

Posts a signal to a specified kernel-only thread.

Syntax

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

void kthread_kill ( tid,  sig)
tid_t tid;
int sig;
```

Parameters

Item	Description
tid	Specifies the target kernel-only thread. If its value is -1, the signal is posted to the calling thread.
sig	Specifies the signal number to post.

Description

The **kthread_kill** kernel service posts the signal *sig* to the kernel thread specified by the *tid* parameter. When the service is called from the process environment, the target thread must be in the same process as the calling thread. When the service is called from the interrupt environment, the signal is posted to the target thread, without a permission check.

Execution Environment

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

Return Values

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

Related reference:

"sig_chk Kernel Service" on page 472

Related information:

Process and Exception Management Kernel Services

kthread_start Kernel Service

Purpose

Starts a previously created kernel-only thread.

Syntax

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

int kthread_start ( tid,  i_func,  i_data_addr,  i_data_len,  i_stackaddr,
i_sigmask)
tid_t tid;
int (*i_func) (void *);
void *i_data_addr;
size_t i_data_len;
void *i_stackaddr;
sigset_t *i_sigmask;
```

Parameters

Item	Description
tid	Specifies the kernel-only thread to start.
i_func	Points to the entry-point routine of the kernel-only thread.
i_data_addr	Points to data that will be passed to the entry-point routine.
i_data_len	Specifies the length of the data chunk.
i_stackaddr	Specifies the stack's base address for the kernel-only thread.
i_sigmask	Specifies the set of signal to block from delivery when the new kernel-only thread begins execution.

Description

The **kthread_start** kernel service starts the kernel-only thread specified by the *tid* parameter. The thread must have been previously created with the **thread_create** kernel service, and its state must be **TSIDL**.

This kernel service initializes and schedules the thread for the processor. Its state is changed to **TSRUN**. The thread is initialized so that it begins executing at the entry point specified by the *i_func* parameter, and that the signals specified by the *i_sigmask* parameter are blocked from delivery.

The thread's entry point gets one parameter, a pointer to a chunk of data that is copied to the base of the thread's stack. The *i_data_addr* and *i_data_len* parameters specify the location and quantity of data to copy. The format of the data must be agreed upon by the initializing and initialized thread.

The thread's stack's base address is specified by the *i_stackaddr* parameter. If a value of zero is specified, the kernel will allocate the memory for the stack (96K). This memory will be reclaimed by the system when the thread terminates. If a non-zero value is specified, then the caller should allocate the backing memory for the stack. Since stacks grow from high addresses to lower addresses, the *i_stackaddr* parameter specifies the highest address for the thread's stack.

The thread will be automatically terminated when it returns from the entry point routine. If it is the last thread in the process, then the process will be exited.

Execution Environment

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

Return Values

The **kthread_start** kernel service returns one of the following values:

Item	Description
0	Indicates a successful start.
ESRCH	Indicates that the *tid* parameter is not valid.

Related reference:
"thread_create Kernel Service" on page 484
Related information:
Process and Exception Management Kernel Services

kvmgetinfo Kernel Service

Purpose

Retrieves Virtual Memory Manager (VMM) information.

Syntax

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

int kvmgetinfo ( void *out, int command, int arg)
```

Description

The **kvmgetinfo** kernel service returns the current value of certain VMM parameters.

Parameters

Item	Description
out	Specifies the address where VMM information should be returned.

Item **Description**

command

Specifies which information should be returned. The valid values for the *command* parameter are decribed below:

VMINFO
The content of **vminfo** structure (described in **sys/vminfo.h**) will be returned. The *out* parameter should point to a **vminfo** structure and the *arg* parameter should be the size of this structure. The smaller of the *arg* or *sizeof* (**struct vminfo**) parameters will be copied.

VMINFO_ABRIDGED
The content of the **vminfo** structure (described in the **sys/vminfo.h** file) is returned. For this command, only the non-time consuming statistics are updated, so this command must be used in performance-critical applications rather than the **VMINFO** command. The *out* parameter must point to a **vminfo** structure and the *arg* parameter must be the size of this structure. The smaller of the *arg* or *sizeof* (**struct vminfo**) parameters are copied.

VM_PAGE_INFO
The size, in bytes, of the page backing the address specified in the *addr* field of the **vm_page_info** structure (described in the **sys/vminfo.h** file) is returned. The *out* parameter should point to a **vm_page_info** structure with the *addr* field set to the desired address of which to query the page size. This address, *addr*, is interpreted as an address in the address space of the current running process. The *arg* parameter should be the size of the **vm_page_info** structure.

IPC_LIMITS
The content of the **ipc_limits** struct (described in the **sys/vminfo.h** file) is returned. The *out* parameter should point to an **ipc_limits** structure and *arg* should be the size of this structure. The smaller of the *arg* or *sizeof* (struct **ipc_limits**) parameters will be copied. The **ipc_limits** struct contains the inter-process communication (IPC) limits for the system.

VMINFO_GETPSIZES
Reports a system's supported page sizes. When *arg* is 0, the *out* parameter is ignored, and the number of supported page sizes is returned. When *arg* is greater than 0, *arg* indicates the number of page sizes to report, and *out* must be a pointer to an array with *arg* number of psize_t types. The array of psize_t types is updated with the system's supported page sizes in sorted order starting with the smallest supported page size. The number of array entries updated with page sizes is returned.

VMINFO_PSIZE
Reports detailed VMM statistics for a specified page size. The *out* parameter must point to a **vminfo_psize** structure with the *psize* field set to a page size, in bytes, for which to return statistics. The *arg* parameter should be the size of the **vminfo_psize** structure.

arg An additional parameter that will depend upon the *command* parameter.

Execution Environment

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

Return Values

The following return values apply to all commands other than **VMINFO_GETPSIZES**:

Item	Description
0	Indicates successful completion.
ENOSYS	Indicates the *command* parameter is not valid (or not yet implemented).
EINVAL	When VM_PAGE_INFO is the command, the *adr* field of the **vm_page_info** structure is an invalid address.

When **VMINFO_GETPSIZES** is specified as the command, -1 is returned if the **kvmgetinfo()** kernel service is unsuccessful. Otherwise, the **kvmgetinfo()** kernel service returns a number of page sizes when the **VMINFO_GETPSIZES** command is specified.

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

kwpar_checkpoint_status Kernel Service

Purpose

Provides a method for kernel services to inform the system that an event occurred within a workload partition (WPAR) that denies or subsequently reallows a checkpoint of the WPAR.

Syntax

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

int kwpar_checkpoint_status (*kcid, cmd, varp***)**
cid_t *kcid***;**
int *cmd***;**
void * *varp***;**

Parameters

Item	Description
cmd	An integer command that informs the API what action to take on behalf of the caller.
kcid	The WPAR ID where the command operation is to take place.
varp	A void pointer to different elements that depends on the *cmd* parameter. • If the *cmd* parameter is set to the **WPAR_CHECKPOINT_TRY** value, the *varp* parameter is a pointer to an integer variable that contains the number of seconds that the caller is willing to wait before a blocking event is removed. • If the *cmd* parameter is set to the **WPAR_CHECKPOINT_DENY** value, the *varp* parameter is a pointer to a null terminated character string that contains a user readable reason for posting the event.

Cmd Types

The *cmd* parameter is supplied on input to the **kwpar_checkpoint_status** API and describes the type of action or event notification the caller is expecting. The following *cmd* types are supported:

Item	Description
WPAR_CHECKPOINT_DENY	The caller is experiencing an event within the WPAR identified by the *kcid* parameter that would deny a checkpoint operation. The caller must supply a pointer to a user readable character string in the *varp* parameter.
WPAR_CHECKPOINT_ALLOW	The caller is clearing a previous checkpoint denial operation. Deny and allow operations are cumulative and thus each denial operation must be matched with an allow operation before a checkpoint is finally reallowed.
WPAR_CHECKPOINT_TRY	Used by the AIX checkpoint system itself. The caller supplies the *varp* pointer to an integer that contains a “willing to wait” timeout in seconds before a checkpoint denial operation is cleared.
WPAR_CHECKPOINT_CLEAR	Used by the AIX checkpoint system itself. The caller completed a checkpoint after a successful **WPAR_TRY_CHKPNT** operation.
WPAR_RESTART_CLEAR	Used by the AIX checkpoint system itself. The caller completed a restart. The WPAR restart state is initially set when the WPAR is re-created on the arrival system.

Description

The **kwpar_checkpoint_status** kernel service provides a mechanism for kernel services to inform or query the system about a checkpoint denial event. Kernel extensions that experience a temporary event which prevents a WPAR from being the target of a checkpoint operation, must use this API to deny and then to subsequently reallow a checkpoint when the event clears. An example denial event might occur if a device open is in an unserialized interim state that cannot handle a checkpoint operation.

Execution Environment

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

Return Values

Item	Description
0	Success.
non-zero	Failure.

Error Codes

The **kwpar_checkpoint_status** service fails if one or more of the following errors occur:

Item	Description
EINVAL	The caller supplied an invalid *cmd* or other parameter.
ENOENT	No WPAR with the *kcid* ID is active in the system.
EBUSY	Either of the following situations can lead to the **EBUSY** error. • WPAR is in a checkpoint or restart state. The caller is unsuccessful in a **WPAR_CHECKPOINT_DENY** operation. • WPAR is in a state that cannot participate in a checkpoint. The caller is unsuccessful in a **WPAR_CHECKPOINT_TRY** operation.
ETIMEDOUT	The caller is waiting for a timeout period during a **WPAR_CHECKPOINT_TRY** operation but the timer expired.

Related reference:

"WPAR_CKPT_QUERY (Checkpoint Query) Device Driver ioctl Operation" on page 583

kwpar_err Kernel Service

Purpose

Logs an error message for a given Workload Partition.

Syntax

```
int kwpar_err(kcid,cat_file_name,msg_set_no,msg_no,default_fmt_msg)
cid_t kcid;
char* cat_file_name;
unsigned int msg_set_no;
unsigned int msg_no;
char* default_fmt_msg;
```

Description

The **kwpar_err** interface provides a mechanism to log error messages for a given WPAR from a kernel routine. Each WPAR can hold up to 1 KB of error messages. If there is enough space to log the new message, the command logs the message; otherwise, it fails. The **kwpar_err** routine is pinned and as such can be called from the interrupt handlers as well.

Parameter

Item	Description
kcid	Specified the **cid** of the WPAR.
cat_file_name	Specifies the catalog file name to be used for translation.
msg_set_no	Specifies the message set number of the error message in the catalog file.
msg_no	Specifies the message number of the error message.
default_fmt_msg	Specifies the default message string. Follows the same syntax as the **printf** subroutine *Format* parameter. Floating point is not supported.
...	Specifies the arguments to the message if any.

Return values

Item	Description
0	Success
-1	Failure

Error codes

Item	Description
ENOMEM	Not enough memory
EINVAL	Invalid parameter

Example

```
To log an error message into WPAR with cid 4, enter

kwpar_err(4, "wparerrs.cat",1,10,"%s : command failed", "mycommand");
...
```

Related information:

wpar_log_err subroutine

wpar_print_err subroutine

wparerr command

kwpar_getname Kernel Service

Purpose

Returns the workload partition name associated with the requested ID.

Syntax

```
#include<sys/wparid.h>
#include<sys/xmem.h>

int kwpar_getname(kcid, buffer, length, adspace)
cid_t kcid;
char * buffer;
size_t length;
int adspace;
```

Description

Get the name associated with the workload partition ID (*kcid*) and write it to the output buffer. The maximum number of bytes to write is limited by the *length* parameter. The *length* parameter cannot exceed MAXCORRALNAMELEN. The service writes to either user space or kernel space, depending on the value specified for the *adspace* parameter.

Parameters

Item	Description
kcid	Specifies the workload partition ID.
buffer	Points to the buffer where the workload partition name is stored.
length	Specifies the maximum number of bytes to return.
adspace	Indicates in which part of memory the buffer parameter is located: **SYS_ADSPACE** Indicates that the *buffer* parameter is in the kernel memory. **USER_ADSPACE** Indicates that the *buffer* parameter is in the application memory.

Execution Environment

Process environment only.

Return Values

Item	Description
0	The command completed successfully.
EINVAL	Invalid WPAR ID or specified length is greater than MAXCORRALNAMELEN.
EFAULT	Error during copyout to user space.

kwpar_getrootpath Kernel Service

Purpose

Returns the root path of the workload partition associated with the requested ID.

Syntax

#include<sys/wparid.h>

```
int kwpar_getrootpath(kcid, length,  buffer)
cid_t kcid;
size_t * length;
char * buffer;
```

Description

Get the root path of the workload partition associated with the *kcid* parameter and copy it to the output buffer. On entry, the value specified for the *length* parameter indicates the size of the output buffer. On return, the value specified for the *length* parameter, contains the size of the root path. If the value for the *length* parameter on entry is smaller than the actual path length, then **ENOSPC** is returned. Then, the *length* parameter is set to the actual length of the root path.

Parameters

Item	Description
kcid	Specifies the workload partition ID.
length	Specifies the maximum number of bytes to return.
buffer	Points to the buffer where the workload partition root path will be stored.

Execution Environment

Process environment only.

Return Values

Item	Description
0	The command completed successfully.
EINVAL	Error indicating that *buffer* is NULL, *length* is NULL, or **length* is 0.
ENOENT	Invalid WPAR ID specified for the *kcid* parameter.
ENOSPC	Insufficient space in *buffer* to copy path.

kwpar_isappwpar Kernel Service

Purpose

Returns whether a workload partition is an application workload partition.

Syntax

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

int kwpar_isappwpar(kcid)
cid_t kcid;
```

Description

Checks whether the workload partition associated with the *kcid* is an application workload partition.

Parameters

Item	Description
kcid	Specifies the workload partition ID.

Execution Environment

Process environment only.

Return Values

Item	Description
1	Workload partition is an application workload partition.
0	Workload partition is not an application workload partition.
-1	Indicates that the command did not complete successfully.

kwpar_r2vmap_devno Kernel Service

Purpose

Maps a real device number to the corresponding virtual device number for a given workload partition (WPAR).

Syntax

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

int kwpar_r2vmap_devno (*wparid*, *vdevno*, *rdevno***)**
cid_t *wparid*;
dev_t *rdevno*;
dev_t * *vdevno*;

Parameters

Item	Description
wparid	WPAR identifier. This parameter is required.
rdevno	Real device number. This parameter is required.
vdevno	Points to the data area that will contain the virtual device number. This parameter is passed by reference. This parameter is optional.

Description

The **kwpar_r2vmap_devno** kernel service provides the ability to translate a real device number, maintained in the kernel device switch table, to the corresponding virtual device number maintained in the user space. The caller must specify an existing WPAR identifier with the *wparid* parameter and a valid real device number with the *rdevno* parameter. The **kwpar_r2vmap_devno** kernel service writes the corresponding virtual device number to the data area pointed to by the *vdevno* parameter (if specified). If the *vdevno* parameter is not specified, the return code indicates whether a mapping exists for the given WPAR identifier and real device number.

A mapping for the specified virtual device number must exist for the **kwpar_v2rmap_devno** kernel service to succeed.

Execution Environment

The **kwpar_r2vmap_devno** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Success.
non-zero	Failure.

Error Codes

The **kwpar_r2vmap_devno** service fails if one or more of the following errors occur:

Item	Description
EINVAL	Either the *wparid* or *rdevno* argument is invalid.
ENXIO	Unable to locate the WPAR device map associated with the given WPAR ID.
ESRCH	Unable to locate a mapping for the given real device number *rdevno*.

Related reference:

"kwpar_v2rmap_devno Kernel Service" on page 325

"kwpar_regdevno Kernel Service" on page 320

"kwpar_unregdevno Kernel Service" on page 323

kwpar_r2vmap_pid Kernel Service

Purpose

Maps a real process ID to the equivalent virtual process ID assigned within a workload partition.

Syntax

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

pid_t kwpar_r2vmap_pid (*kcidp*, *rpid***)**
cid_t * *kcidp***;**
pid_t *rpid***;**

Parameters

Item	Description
kcidp	A pointer to a memory location where the workload partition (WPAR) ID associated with the *rpid* parameter is returned.
rpid	The real process ID on which to translate a real process ID to a virtual process ID.

Description

The **kwpar_r2vmap_pid** kernel service provides a mapping from a real process ID to a virtual process ID assigned within the workload partition. In most instances, the real and virtual process IDs are the same except in cases where the Workload Partition Mobility is in effect or for certain system services such as the **init** command which always have different real and virtual process IDs.

Usually kernel services dealing with process IDs only accept real process IDs. However, in some instances it might be necessary for kernel extensions, which communicate with other WPAR services or with processes within the WPAR, to know and communicate with virtual process IDs.

Execution Environment

The **kwpar_r2vmap_pid** kernel service can be called from the process environment only.

Return Values

If the **kwpar_r2vmap_pid** kernel service succeeds, it returns the virtual *pid_t* value associated with the *rpid* value provided on input. If the kernel service fails or if there is no virtual process ID associated with the *rpid* value, the *rpid* value is returned.

Related reference:

"kwpar_v2rmap_pid Kernel Service" on page 326

kwpar_r2vmap_tid Kernel Service

Purpose

Maps a real thread ID to the equivalent virtual thread ID assigned within a workload partition.

Syntax

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

tid_t kwpar_r2vmap_tid (*kcidp*, *rtid***)**
cid_t * *kcidp***;**
tid_t *rtid***;**

Parameters

Item	Description
kcidp	A pointer to a memory location where the WPAR ID associated with the *rtid* parameter is returned.
rtid	The real thread ID on which to translate a real process ID to a virtual process ID.

Description

The **kwpar_r2vmap_tid** kernel service provides a mapping from a real thread ID to a virtual thread ID assigned within the workload partition. In most instances, the real and virtual thread IDs are the same except in cases where the Workload Partition Mobility is in effect.

Normally kernel services dealing with thread IDs accept only real thread IDs. However, in some instances it might be necessary for kernel extensions, which communicate with other WPAR services or with processes within the WPAR, to know and communicate with virtual thread IDs.

Execution Environment

The **kwpar_r2vmap_tid** kernel service can be called from the process environment only.

Return Values

If the **kwpar_r2vmap_tid** kernel service succeeds, it returns the virtual *tid_t* value associated with the *rtid* value provided on input. If the kernel service fails or if there is no virtual process ID associated with the *rtid* value, the *rtid* value is returned.

Related reference:

"kwpar_v2rmap_tid Kernel Service" on page 327

kwpar_regdevno Kernel Service

Purpose

Registers a virtual device number for a given workload partition (WPAR) by mapping it to a real device number in the device switch table.

Syntax

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

int kwpar_regdevno (*wparid*, *vdevno*, *rdevno***)**
cid_t *wparid***;**
dev_t *vdevno***;**
dev_t * *rdevno***;**

Parameters

Item	Description
wparid	WPAR ID. This parameter is required.
vdevno	Virtual device number. This parameter is required.
rdevno	Points to the data area that will contains the real device number. This parameter is passed by reference. This parameter is required.

Description

The **kwpar_regdevno** kernel service provides the ability to register a virtual device number for a given WPAR by mapping it to a real device number in the device switch table. The **kwpar_regdevno** kernel service performs the following steps:

1. Locates a free slot in the kernel device switch table and reserves it for the WPAR specified by the *wparid* parameter.
2. Creates a mapping between the virtual device number, which is specified by the *vdevno* parameter, to the real device number reserved in the previous step.
3. The newly reserved real device number is passed back to the caller through the *rdevno* parameter.

Execution Environment

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

Return Values

Item	Description
0	Success.
non-zero	Failure.

Error Codes

The **kwpar_regdevno** kernel service fails if one or more of the following errors occur:

Item	Description
EINVAL	Either the *wparid* or *vdevno* argument is not valid.
ENXIO	Unable to locate the WPAR device map associated with the given WPAR ID.
ENOTEMPTY	The virtual device number *vdevno* is already mapped.

Related reference:

"kwpar_r2vmap_devno Kernel Service" on page 317

"kwpar_v2rmap_devno Kernel Service" on page 325

"kwpar_unregdevno Kernel Service" on page 323

kwpar_reghook Kernel Service

Purpose

Registers a function callback with workload partition (WPAR) kernel services. Callback functions are subsequently performed when specific WPAR conditions occur.

Syntax

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

regkey_t kwpar_reghook (*hooktype*, *hookp***)**
int *hooktype***;**
void * *hookp***;**

Parameters

Item	Description
hooktype	Identifies the form of the *hookp* pointer.
hookp	A pointer to a memory location that might contain function pointers or other structure elements that are interpreted depending on the supplied *hooktype* value.

Hook Types

The *hooktype* parameter is supplied on input to the **kwpar_reghook** return and describes the form of the second parameter. The supported hook types are as follows:

Item	Description
WPAR_NOTIFY_HOOK	Identifies the form of the *hookp* parameter as being of type **wpar_config_hook_t**.

The **wpar_config_hook_t** structure contains the following fields:

Item	Description
uint current_hiwater	On output from the **kwpar_reghook** service, this field contains the current upper number of WPARs that became active on this boot instance of the AIX operating system. WPAR IDs are allocated in numeric order. Kernel subsystems that want to size internal components according to the number of active WPARs must register a **WPAR_NOTIFY_HOOK** hook type and examine the **current_hiwater** value for existing WPARs during registration. Future WPAR activation after hook registration calls the specified **configp** function within the **wpar_config_hook_t** element. See the **WPARSTART** flags later in this section for a further description of the WPAR activation.
wpar_config_func_t configp	On input, this field contains a pointer to a callback routine that is started by the WPAR kernel services during the activation and the deactivation of workload partitions within the AIX kernel.

The syntax for the **wpar_config_func_t** is as follows:

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

typedef int * wpar_config_func_t (*flags*, *cid*, *corralp*, *unused***)**
int *flags***;**
cid_t *cid***;**
struct corral * *corralp***;**
void * *unused***;**

The parameters are as follows:

Item	Description
flags	Information regarding the type of condition that is occurring within the workload partition.
cid	The ID for the workload partition experiencing the condition.
corralp	A pointer to a kernel copy of the **corral** structure that might be supplied from the user space at the start of the condition processing.
unused	Currently unused and must be set to NULL. It might be expanded to contain more information in later revisions of this API.

The *flags* parameter can have the following potential values:

Item	Description
WPARSTART	Signifies that the WPAR is undergoing activation. The callout to registered routines occurs before any other kernel subsystem processing occurs. Kernel components registering and desiring to see the WPAR activation are informed that a new WPAR with the *cid* parameter set is going to enter the AIX kernel system.
WPARSTOP	Signifies that the WPAR underwent deactivation. The callout to registered routines occurs after all other kernel subsystem processing occurs. Kernel components registering and desiring to see the WPAR deactivation are informed that an existing WPAR with the *cid* parameter set left the AIX kernel system.

Description

The **kwpar_reghook** kernel service provides a mechanism for other kernel services to register callbacks and retrieve information when certain workload partition conditions occur.

Execution Environment

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

Return Values

If the **kwpar_reghook** kernel service is successful, it returns a registration key that can subsequently be used with the **kwpar_unreghook** kernel service. If the kernel service fails, it returns a numeric value equivalent to the BADREGKEY definition found in the **wparid.h** file.

Error Codes

The **kwpar_reghook** kernel service fails if no space remains to record additional registration hook.

Related reference:

"kwpar_unreghook Kernel Service" on page 324

kwpar_unregdevno Kernel Service

Purpose

Unregisters the mapping associated with a real device number for a given workload partition (WPAR).

Syntax

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

int kwpar_unregdevno (*wparid, rdevno***)**
cid_t *wparid***;**
dev_t *rdevno***;**

Parameters

Item	**Description**
wparid	WPAR identifier. This parameter is required.
rdevno	Real device number. This parameter is required.

Description

The **kwpar_unregdevno** kernel service provides the ability to unregister the mapping associated with a real device number for a given WPAR. The **kwpar_unregdevno** kernel service will perform the following steps:

1. Deletes the virtual-to-real mapping associated with the real device number specified by the *rdevno* parameter for the WPAR specified by the *wparid* parameter.
2. Releases the reserve associated with the real device number specified by the *rdevno* parameter.

Execution Environment

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

Return Values

Item	**Description**
0	Success.
non-zero	Failure.

Error Codes

The **kwpar_unregdevno** kernel service fails if one or more of the following errors occur:

Item	Description
EINVAL	Either the *wparid* or *rdevno* argument is not valid.
ENXIO	Unable to locate the WPAR device map associated with the given WPAR ID.
ESRCH	Unable to locate the mapping for the given real device number *rdevno*.

Related reference:

"kwpar_r2vmap_devno Kernel Service" on page 317

"kwpar_v2rmap_devno Kernel Service" on page 325

"kwpar_regdevno Kernel Service" on page 320

kwpar_unreghook Kernel Service

Purpose

Removes a previously registered workload partition (WPAR) callback hook.

Syntax

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

int kwpar_unreghook (*key*)
regkey_t *key*;

Parameters

Item	Description
key	The registration key of the hook that the caller wants to un-register. This key is equivalent to the key returned from a hook registration with the **kwpar_reghook** kernel service.

Description

The **kwpar_unreghook** kernel service informs workload partitions that the caller no longer wants to receive callouts for WPAR conditions.

Execution Environment

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

Return Values

Item	Description
0	Success.
non-zero	Failure.

Error Codes

The **kwpar_unreghook** service fails if one or more of the following errors occur:

Item	Description
EINVAL	Not a valid registration key.
EPERM	Not allowed to un-register this key.

Related reference:

"kwpar_reghook Kernel Service" on page 321

kwpar_v2rmap_devno Kernel Service

Purpose

Maps a virtual device number to the corresponding real device number in the device switch table for a given workload partition (WPAR).

Syntax

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

int kwpar_v2rmap_devno (*wparid, vdevno, rdevno***)**
cid_t *wparid***;**
dev_t *vdevno***;**
dev_t * *rdevno***;**

Parameters

Item	Description
wparid	WPAR identifier. This parameter is required.
vdevno	Virtual device number. This parameter is required.
rdevno	Points to the data area that will contain the real device number. This parameter is passed by reference. This parameter is optional.

Description

The **kwpar_v2rmap_devno** kernel service provides the ability to translate a virtual device number maintained in user space to the corresponding real device number maintained in the kernel device switch table. The caller must specify an existing WPAR identifier with the *wparid* parameter and a valid virtual device number with the *vdevno* parameter. The **kwpar_v2rmap_devno** kernel service will write the corresponding real device number to the data area pointed to by the *rdevno* parameter if it is specified. If the *rdevno* parameter is not specified, the return code will indicate whether a mapping exists for the given WPAR identifier and virtual device number.

A mapping for the specified virtual device number must exist for the **kwpar_v2rmap_devno** kernel service to succeed.

Execution Environment

The **kwpar_v2rmap_devno** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Success.
non-zero	Failure.

Error Codes

The **kwpar_v2rmap_devno** service fails if one or more of the following errors occur:

Item	Description
EINVAL	Either the *wparid* or *vdevno* argument is not valid.
ENXIO	Unable to locate the WPAR device map associated with the given WPAR id.
ENODEV	Unable to locate the mapping for the given virtual device number.

Related reference:

"kwpar_r2vmap_devno Kernel Service" on page 317

"kwpar_regdevno Kernel Service" on page 320

"kwpar_unregdevno Kernel Service" on page 323

kwpar_v2rmap_pid Kernel Service

Purpose

Maps a virtual process ID associated with a process within a workload partition to the equivalent real process ID.

Syntax

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

pid_t kwpar_v2rmap_pid (*kcid*, *vpid***)**
cid_t *kcid***;**
pid_t *vpid***;**

Parameters

Item	Description
kcid	The workload partition (WPAR) ID associated with the *vpid* parameter. Equivalent virtual process IDs can be in use across different processes in different WPARs. Thus the caller must provide the WPAR ID for which a virtual to real mapping is to occur.
vpid	The virtual process ID on which to perform a virtual to real mapping.

Description

The **kwpar_v2rmap_pid** kernel service provides a mapping from a virtual process ID associated with a process in a workload partition to the equivalent real process ID. In most instances, both the real and virtual process IDs are the same, except in cases where the Workload Partition Mobility is in effect.

Normally, kernel services dealing with process IDs accept only real thread IDs. In some instances where a kernel extension is communicating with other WPAR services or with processes within the WPAR, a mapping from virtual to real process IDs might be needed.

Execution Environment

The **kwpar_v2rmap_pid** kernel service can be called from the process environment only.

Return Values

If the **kwpar_v2rmap_pid** kernel service succeeds, it returns the real *pid_t* value associated with the *vpid* value provided on input. If the kernel service fails, or if there is no real thread ID associated with the *vpid* value, then the *vpid* value is returned.

Related reference:

"kwpar_r2vmap_pid Kernel Service" on page 318

kwpar_v2rmap_tid Kernel Service

Purpose

Maps a virtual thread ID associated with a thread within a workload partition to the equivalent real thread ID.

Syntax

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

tid_t kwpar_v2rmap_tid (*kcid*, *vtid***)**
cid_t *kcid***;**
tid_t *vtid***;**

Parameters

Item	Description
kcid	The workload partition (WPAR) ID associated with the *vtid* parameter. Equivalent virtual thread IDs can be in use across different threads in different WPARs. Thus the caller must provide the WPAR ID for which a virtual to real mapping is to occur.
vtid	The virtual thread ID on which to perform a virtual to real mapping.

Description

The **kwpar_v2rmap_tid** kernel service provides a mapping from a virtual thread ID associated with a thread in a workload partition to the equivalent real thread ID. In most instances, both the real and virtual thread IDs are the same, except in cases where the Workload Partition Mobility is in effect. Normally, kernel services dealing with thread IDs accept only real thread IDs. In some instances where a kernel extension is communicating with other WPAR services or with processes within the WPAR, a mapping from virtual to real thread IDs might be needed.

Execution Environment

The **kwpar_v2rmap_tid** kernel service can be called from the process environment only.

Return Values

If the **kwpar_v2rmap_tid** kernel service succeeds, it returns the real *tid_t* value associated with the *vtid* value provided on input. If the kernel service fails, or if there is no real thread ID associated with the *vtid* value then the *vtid* value is returned.

Related reference:

"kwpar_r2vmap_tid Kernel Service" on page 319

l

The following kernel services begin with the with the letter l.

ldata_alloc Kernel Service

Purpose

Allocates a pinned storage element from an **ldata** pool.

Syntax

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

void * ldata_alloc (ldatap)

ldata_t ldatap;
```

Description

The **ldata_alloc** kernel service allocates a pinned storage element from a **ldata** pool and returns the address of the element. The **ldata_alloc** kernel service makes a pinned storage element from the **ldata** pool available for use by the caller. The sub-pool from which the element is allocated corresponds to the SRAD on which the call was made. If there are no free pinned elements, a new element cannot be allocated and a NULL value is returned.

After it is allocated, the pinned storage element can be freed to the **ldata** pool through the **ldata_free** kernel service.

Parameters

Item	Description
ldatap	Specifies the handle of the **ldata** pool.

Execution Environment

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

Return Values

Returns a pointer to a pinned storage element allocated from an **ldata** pool or NULL if no element could be allocated.

Implementation Specifics

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

Related reference:

"ldata_create Kernel Service"

"ldata_grow Kernel Service" on page 331

"ldata_free Kernel Service" on page 331

ldata_create Kernel Service

Purpose

Creates a SRAD-aware pinned storage element pool (**ldata** pool) and returns its handle.

Syntax

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

int ldata_create (size, initcount, maxcount, kkey, ldatap)
```

```
size_t size;
long initcount;
long maxcount;
kkey_t kkey;
ldata_t * ldatap;
```

Description

The **ldata_create** kernel service creates a SRAD-aware pool (**ldata** pool) of pinned storage elements, each of the specified size, and returns a handle to the newly-allocated pool. An **ldata** pool consists of a number of sub-pools (one per SRAD). Each sub-pool is physically backed with memory local to its corresponding SRAD. The size of each sub-pool is equal to the value of the *maxcount* parameter multiplied by the value of the *size* parameter. The parameter (*initcount*) specifies the number of pinned storage elements in each sub-pool that should be pre-allocated.

The **ldata** pool can be created with a kernel storage protection key by specifying one through the *kkey* parameter. For compatibility with previous releases, a *kkey* parameter of zero requests no protection. When a protection key is specified, the caller must hold this key when calling any ldata service, including the **ldata_create** kernel service.

After an **ldata** pool is created, its handle can be used to allocate pinned storage elements from the pool through the **ldata_alloc** kernel service and free these elements to the pool through the **ldata_free** kernel services. Elements are allocated and freed to the sub-pool corresponding to the SRAD on which **ldata_alloc** and **ldata_free** are called. If a sub-pool is exhausted of its pinned storage elements, it can be grown by calling the **ldata_grow** kernel service up to *maxcount*.

An **ldata** pool created through the **ldata_create** service can be destroyed by the **ldata_destroy** kernel service.

Parameters

Item	Description
size	Specifies the size, in bytes, of each pinned storage element of the **ldata** pool.
initcount	Specifies the initial count of pinned storage elements, to be contained within the **ldata** pool. Must be a positive integer.
maxcount	Specifies the maximum count of pinned storage elements that can be contained with the **ldata** pool. The value of *maxcount* must be positive and greater than or equal to the value of *initcount*.
kkey	Specifies the kernel storage protection key to be applied to the newly created **ldata** pool. The value must be a valid kernel key number, or zero to indicate that storage protection is not requested.
ldatap	Specifies an address to be set on successful completion with the handle for the newly created **ldata** pool.

Execution Environment

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

Return Values

Item	Description
0	Completed successfully. The handle for **ldata** storage is returned in *ldatap*.
EINVAL	Invalid input parameters given. Invalid *initcount*, *maxcount* or *kkey*. The *ldatap* parameter is undefined.
ENOMEM	Error encountered. Insufficient memory to satisfy request. The *ldatap* parameter is undefined.

Implementation Specifics

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

Related reference:

"ldata_destroy Kernel Service"

"ldata_grow Kernel Service" on page 331

"ldata_alloc Kernel Service" on page 328

ldata_destroy Kernel Service

Purpose

Destroys an **ldata** pool created by the **ldata_create** kernel service.

Syntax

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

void ldata_destroy (ldatap)

ldata_t ldatap;
```

Description

The **ldata_destroy** kernel service destroys an **ldata** pool previously created by an **ldata_create** call. This routine assumes that all elements allocated from the pool have been freed back to the pool and there are no longer any active elements in the pool.

The **ldata_destroy** call unpins and frees all of the storage associated with the handle.

Parameters

Item	Description
ldatap	Specifies the handle of the **ldata** pool to be destroyed.

Execution Environment

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

Return Values

None.

Implementation Specifics

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

Related reference:

"ldata_create Kernel Service" on page 328 "ldata_alloc Kernel Service" on page 328

"ldata_free Kernel Service"

ldata_free Kernel Service

Purpose

Frees a storage element that is pinned to an **ldata** pool.

Syntax

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

void ldata_free (ldatap, elementp)

ldata_t ldatap;
void * elementp;
```

Description

The **ldata_free** kernel service frees a pinned storage element that was previously allocated to an **ldata** pool. The pinned storage element is identified through the *elementp* parameter. The element identified by *elementp* is freed to the sub-pool corresponding to the SRAD that allocated the element.

Parameters

Item	**Description**
ldatap	Specifies the handle of the **ldata** pool.
elementp	Specifies the address of the pinned storage element to be freed.

Execution Environment

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

Return Values

None.

Implementation Specifics

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

Related reference:

"ldata_alloc Kernel Service" on page 328

ldata_grow Kernel Service

Purpose

Expands the count of available pinned storage elements contained within an **ldata** pool.

Syntax

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

int ldata_grow (ldatap, count)

ldata_t ldatap;
long count;
```

Description

The **ldata_grow** kernel service increases the number of pinned storage elements contained within a per-SRAD sub-pool associated with the **ldata** handle *ldatap*, by *count*. If the **ldata_alloc** call fails because there are no more free pinned storage elements in a sub-pool, use the **ldata_grow** kernel service. The **ldata_grow** kernel service pins additional count elements from the sub-pool and makes them available for the **ldata_alloc** call. All of the sub-pools associated with the handle are grown. If count elements are not available or there is not enough pinned memory available, the **ldata_grow** kernel service fails.

Parameters

Item	Description
ldatap	Specifies the handle of the **ldata** pool.
count	Specifies the additional number of storage elements to be pinned in the sub-pool. The *count* value should be greater than 0 and should not increase the sub-pool size beyond the value of *maxcount* specified with the **ldata_create** call.

Execution Environment

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

Return Values

Item	Description
0	Success.
-1	Error encountered. Illegal parameters or insufficient resources.

Implementation Specifics

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

Related reference:

"ldata_create Kernel Service" on page 328

ldmp_bufest, ldmp_timeleft, ldmp_xmalloc, ldmp_xmfree, and ldmp_errstr Kernel Services

Purpose

Obtains information about the current live dump.

Syntax

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

kerrno_t ldmp_bufest (id, cb, len)
dumpid_t id;
ras_block_t cb;
size_t *len;

kerrno_t ldmp_timeleft (id, timeleft)
dumpid_t id;
long *timeleft;

kerrno_t ldmp_xmalloc (id, size, align, p)
dumpid_t id;
```

```
size_t size;
uint align;
void **p;

kerrno_t ldmp_xmfree (id, p)
dumpid_t id;
void *p;

kerrno_t ldmp_errstr (id, cb, str)
dumpid_t id;
ras_block_t cb;
char *str;
```

Parameters

Item	Description
align	Specifies the log base 2 of the desired alignment. The maximum allowed alignment is 12, 4096 byte alignment.
cb	Specifies the ras_block_t for the component.
id	Specifies the ID of the dump.
len	Specifies the estimate of data in bytes that can still be buffered by the specified component in this pass.
p	Specifies the memory block to be allocated or freed.
size	Specifies the memory size to be allocated.
str	Specifies the error message.
timeleft	Specifies the time, in nanoseconds, remaining for this pass. This value only has meaning for a serialized dump. It can be negative.

Description

The **ldmp_bufest** kernel service estimates the number of bytes of dump buffer storage available to this component.

The **ldmp_timeleft** kernel service estimates the time, in nanoseconds, remaining in this pass.

The **ldmp_xmalloc** kernel service allocates storage from the live dump heap.

The **ldmp_xmfree** kernel service frees live dump heap storage.

The **ldmp_errstr** kernel service records an error to be part of the live dump status reporting. The string is contained in the live dump and reported in the error log entry if there is sufficient space.

Important: An error log entry has a maximum length of 2048 bytes. The error string is limited to 128 bytes, including the trailing NULL, and is truncated if too long. The component's path name is also logged.

Tip: The **ldmp_errstr** kernel service can be called multiple times to report multiple errors. Components are encouraged to limit the size of error strings due to limited space in the error log entry.

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_RAS_xxx_BADARGS	Indicates that the arguments for the service are not valid.
EFAULT_RAS_xxx_BADARGS	Indicates that an address argument is not a valid address.
ENOMEM_RAS_LDMP_XMALLOC	Indicates that there is insufficient space in the live dump heap to satisfy this request.

Related reference:

"livedump Kernel Service" on page 336

ldmp_freeparms Kernel Service

Purpose

Frees any data allocated by the live dump associated with an unused **ldmp_parms_t** data item.

Syntax

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

kerrno_t ldmp_freeparms (parms)
ldmp_parms_t *parms;
```

Parameters

Item	Description
parms	Points to an item of **ldmp_parms_t** type.

Description

The **ldmp_freeparms** kernel service is used in the event that you have partially set up the **ldmp_parms_t** data item, but do not want to take a dump. You can use the **ldmp_freeparms** kernel service to clean up any data allocated by the live dump subsystem. However, you can always call the **ldmp_freeparms** kernel service after the **livedump** kernel service, and the **ldmp_freeparms** kernel service returns normally if there is nothing to free.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_RAS_LDMP_FREEPARMS	Indicates that the area is not a valid **ldmp_parms_t** data area.
EFAULT_RAS_LDMP_FREEPARMS	Indicates that a memory fault results.

Related reference:

"ldmp_setupparms Kernel Service"

"livedump Kernel Service" on page 336

ldmp_setupparms Kernel Service

Purpose

Sets up the **ldmp_parms_t** parameter for the **livedump** kernel service.

Syntax

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

kerrno_t ldmp_setupparms (parms)
ldmp_parms_t *parms;
```

Parameters

Item	Description
parms	Points to an item of **ldmp_parms_t** type.

Description

The **ldmp_setupparms** kernel service simplifies the process of setting up a live dump by setting up the **ldmp_parms_t** parameter. It does not allocate any storage.

The **ldmp_setupparms** kernel service performs the following setup for the **ldmp_parms_t** parameter:

Item	Description
Field	**Value**
ldp_eyec	eyecatcher for ldmp_parms
ldp_vers	current version
ldp_flags	0
ldp_prio	LDPP_CRITICAL
ldp_recov	NULL
ldp_func	NULL
ldp_namepref	NULL
ldp_errcode	0
ldp_symptom	NULL
ldp_title	NULL
ldp_rsvd1	NULL

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful completion.
EFAULT_RAS_LDMP_SETUPPARMS	Indicates that the address is not valid.

Related reference:

"livedump Kernel Service" on page 336

limit_sigs or sigsetmask Kernel Service

Purpose

Changes the signal mask for the calling kernel thread.

Syntax

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

void limit_sigs (
 siglist,
 old_mask)
sigset_t *siglist;
sigset_t *old_mask;

void sigsetmask ( old_mask)
sigset_t *old_mask;
```

Parameters

Item	Description
siglist	Specifies the signal set to deliver.
old_mask	Points to the old signal set.

Description

The **limit_sigs** kernel service changes the signal mask for the calling kernel thread such that only the signals specified by the *siglist* parameter will be delivered, unless they are currently being blocked or ignored.

The old signal mask is returned via the *old_mask* parameter. If the *siglist* parameter is **NULL**, the signal mask is not changed; it can be used for getting the current signal mask.

The **sigsetmask** kernel service should be used to restore the set of blocked signals for the calling thread. The typical usage of these services is the following:

```
sigset_t allowed = limited set of signals
sigset_t old;

/* limits the set of delivered signals */
limit_sigs (&allowed, &old);

   /* do something with a limited set of delivered signals */

/* restore the original set */
sigsetmask (&old);
```

Execution Environment

The **limit_sigs** and **sigsetmask** kernel services can be called from the process environment only.

Return Values

The **limit_sigs** and **sigsetmask** kernel services have no return values.

Related reference:

"kthread_kill Kernel Service" on page 309

Related information:

Process and Exception Management Kernel Services

livedump Kernel Service

Purpose

Starts a live dump.

Syntax

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

```
kerrno_t livedump (parms)
ldmp_parms_t *parms;
```

Parameters

Item	Description
parms	Points to an item of **ldmp_parms_t** type.

Description

The **livedump** kernel service initiates a live dump. It can be called from either the kernel or a kernel extension. Storage associated with the dump is not entirely freed until the dump has been written to disk, or the **livedump** kernel service returns an error indicating the dump was not taken.

Execution Environment

The **livedump** kernel service can be called from either the process or interrupt environment. Only a serialized, synchronous dump can be started from the interrupt level, and the dump is limited to one pass.

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_RAS_LIVEDUMP_PARM	Indicates that one or more parameters are not valid.
EFAULT_RAS_LIVEDUMP_PARM	Indicates that a memory fault occurs.
EINVAL_RAS_LIVEDUMP_COMP	Indicates one or more components are not valid.
EINVAL_RAS_LIVEDUMP_NOCOMPS	Indicates that no valid components were given.

Related reference:

"dmp_compspec and dmp_compext Kernel Services" on page 92

"dmp_eaddr, dmp_context, dmp_tid, dmp_pid, dmp_errbuf, dmp_mtrc, dmp_systrace, and dmp_ct Kernel Services" on page 99

lock_alloc Kernel Service

Purpose

Allocates system memory for a simple or complex lock.

Syntax

```
#include <sys/lock_def.h>
#include <sys/lock_alloc.h>

void lock_alloc ( lock_addr,  flags,  class,  occurrence)
void *lock_addr;
int flags;
short class;
short occurrence;
```

Parameters

Item	Description
lock_addr	Specifies a valid simple or complex lock address.
flags	Specifies whether the memory allocated is to be pinned or pageable. Set this parameter as follows: **LOCK_ALLOC_PIN** Allocate pinned memory; use if it is not permissible to take a page fault while calling a locking kernel service for this lock. **LOCK_ALLOC_PAGED** Allocate pageable memory; use if it is permissible to take a page fault while calling a locking kernel service for this lock.

Item	Description
class	Specifies the family which the lock belongs to.
occurrence	Identifies the instance of the lock within the family. If only one instance of the lock is defined, this parameter should be set to -1.

Description

The **lock_alloc** kernel service allocates system memory for a simple or complex lock. The **lock_alloc** kernel service must be called for each simple or complex before the lock is initialized and used. The memory allocated is for internal lock instrumentation use, and is not returned to the caller; no memory is allocated if instrumentation is not used.

Execution Environment

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

Return Values

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

Related reference:

"lock_free Kernel Service" on page 340

"lock_init Kernel Service" on page 341

Related information:

Understanding Locking

lock_clear_recursive Kernel Service

Purpose

Prevents a complex lock from being acquired recursively.

Syntax

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

void lock_clear_recursive ( lock_addr)
complex_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word which is no longer to be acquired recursively.

Description

The **lock_clear_recursive** kernel service prevents the specified complex lock from being acquired recursively. The lock must have been made recursive with the **lock_set_recursive** kernel service. The calling thread must hold the specified complex lock in write-exclusive mode.

Execution Environment

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

Return Values

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

Related reference:

"lock_init Kernel Service" on page 341

"lock_done Kernel Service"

Related information:

Locking Kernel Services

lock_done Kernel Service

Purpose

Unlocks a complex lock.

Syntax

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

void lock_done ( lock_addr)
complex_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word to unlock.

Description

The **lock_done** kernel services unlocks a complex lock. The calling kernel thread must hold the lock either in shared-read mode or exclusive-write mode. If one or more kernel threads are waiting to acquire the lock in exclusive-write mode, one of these kernel threads (the one with the highest priority) is made runnable and may compete for the lock. Otherwise, any kernel threads which are waiting to acquire the lock in shared-read mode are made runnable. If there was at least one kernel thread waiting for the lock, the priority of the calling kernel thread is recomputed.

If the lock is held recursively, it is not actually released until the **lock_done** kernel service has been called once for each time that the lock was locked.

Execution Environment

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

Return Values

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

Related reference:

"lock_alloc Kernel Service" on page 337

"lock_free Kernel Service"

"lock_init Kernel Service" on page 341

Related information:

Understanding Locking

Locking Kernel Services

lock_free Kernel Service

Purpose

Frees the memory of a simple or complex lock.

Syntax

```
#include <sys/lock_def.h>
#include <sys/lock_alloc.h>

void lock_free ( lock_addr)
void *lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word whose memory is to be freed.

Description

The **lock_free** kernel service frees the memory of a simple or complex lock. The memory freed is the internal operating system memory which was allocated with the **lock_alloc** kernel service.

Note: It is only necessary to call the **lock_free** kernel service when the memory that the corresponding lock was protecting is released. For example, if you allocate memory for an i-node which is to be protected by a lock, you must allocate and initialize the lock before using it. The memory may be used with several i-nodes, each taken from, and returned to, the free i-node pool; the **lock_init** kernel service must be called each time this is done.The **lock_free** kernel service must be called when the memory allocated for the inode is finally freed.

Execution Environment

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

Return Values

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

Related reference:

"lock_alloc Kernel Service" on page 337

Related information:

Understanding Locking

Locking Kernel Services

lock_init Kernel Service

Purpose

Initializes a complex lock.

Syntax

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

void lock_init ( lock_addr,  can_sleep)
complex_lock_t lock_addr;
boolean_t can_sleep;
```

Parameters

Item	Description
lock_addr	Specifies the address of the lock word.
can_sleep	This parameter is ignored.

Description

The **lock_init** kernel service initializes the specified complex lock. This kernel service must be called for each complex lock before the lock is used. The complex lock must previously have been allocated with the **lock_alloc** kernel service. The *can_sleep* parameter is included for compatibility with OSF/1 1.1, but is ignored. Using a value of **TRUE** for this parameter will maintain OSF/1 1.1 semantics.

Execution Environment

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

Return Values

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

Related reference:

"lock_alloc Kernel Service" on page 337

"lock_free Kernel Service" on page 340

Related information:

Understanding Locking

Locking Kernel Services

lock_islocked Kernel Service

Purpose

Tests whether a complex lock is locked.

Syntax

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

int lock_islocked ( lock_addr)
complex_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word to test.

Description

The **lock_islocked** kernel service determines whether the specified complex lock is free, or is locked in either shared-read or exclusive-write mode.

Execution Environment

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

Return Values

Item	Description
TRUE	Indicates that the lock was locked.
FALSE	Indicates that the lock was free.

Related reference:

"lock_init Kernel Service" on page 341

Related information:

Understanding Locking

Locking Kernel Services

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

Item	Description
lock _word	Specifies the address of the lock word.

Item	Description
flags	Specifies the flags that control waiting for a lock. The *flags* parameter is used to control how signals affect waiting for a lock. The four flags are: **LOCK_NDELAY** Controls whether the caller waits for the lock. Setting the flag causes the request to be terminated. The lock is assigned to the caller. Not setting the flag causes the caller to wait until the lock is not owned by another process before the lock is assigned to the caller. **LOCK_SHORT** Prevents signals from terminating the wait for the lock. **LOCK_SHORT** is the default flag for the **lockl** Kernel Service. This flag causes non-preemptive sleep. **LOCK_SIGRET** Causes the wait for the lock to be terminated by an unmasked signal. **LOCK_SIGWAKE** Causes the wait for the lock to be terminated by an unmasked signal and control transferred to the return from the last operation by the **setjmpx** kernel service. **Note:** The **LOCK_SIGRET** flag overrides the **LOCK_SIGWAKE** flag.

Description

Note: The **lockl** kernel service is provided for compatibility only and should not be used in new code, which should instead use simple locks or complex locks.

The **lockl** kernel service locks a conventional lock

The lock word can be located in shared memory. It must be in the process's address space when the **lockl** or **unlockl** services are called. The kernel accesses the lock word only while executing under the caller's process.

The *lock_word* parameter is typically part of the data structure that describes the resource managed by the lock. This parameter must be initialized to the **LOCK_AVAIL** value before the first call to the **lockl** service. Only the **lockl** and **unlockl** services can alter this parameter while the lock is in use.

The **lockl** service is nestable. The caller should use the **LOCK_SUCC** value for determining when to call the **unlockl** service to unlock the conventional lock.

The **lockl** service temporarily assigns the owner the process priority of the most favored waiter for the lock.

A process must release all locks before terminating or leaving kernel mode. Signals are not delivered to kernel processes while those processes own any lock. "Understanding System Call Execution" in *Kernel Extensions and Device Support Programming Concepts* discusses how system calls can use the **lockl** service when accessing global data.

Execution Environment

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

Return Values

Item	Description
LOCK_SUCC	Indicates that the process does not already own the lock or the lock is not owned by another process when the *flags* parameter is set to **LOCK_NDELAY**.
LOCK_NEST	Indicates that the process already owns the lock or the lock is not owned by another process when the *flags* parameter is set to **LOCK_NDELAY**.
LOCK_FAIL	Indicates that the lock is owned by another process when the *flags* parameter is set to **LOCK_NDELAY**.
LOCK_SIG	Indicates that the wait is terminated by a signal when the *flags* parameter is set to **LOCK_SIGRET**.

Related reference:

"unlockl Kernel Service" on page 518

Related information:

Understanding Locking

Locking Kernel Services

lock_mine Kernel Service

Purpose

Checks whether a simple or complex lock is owned by the caller.

Syntax

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

boolean_t lock_mine ( lock_addr)
void *lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word to check.

Description

The **lock_mine** kernel service checks whether the specified simple or complex lock is owned by the calling kernel thread. Because a complex lock held in shared-read mode has no owner, the service returns FALSE in this case. This kernel service is provided to assist with debugging.

Execution Environment

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

Return Values

Item	Description
TRUE	Indicates that the calling kernel thread owns the lock.
FALSE	Indicates that the calling kernel thread does not own the lock, or that a complex lock is held in shared-read mode.

Related reference:

"lock_init Kernel Service" on page 341

"lock_read or lock_try_read Kernel Service" on page 345

"lock_write or lock_try_write Kernel Service" on page 347

Related information:

Locking Kernel Services

lock_read or lock_try_read Kernel Service

Purpose

Locks a complex lock in shared-read mode.

Syntax

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

void lock_read ( lock_addr)
complex_lock_t lock_addr;

boolean_t lock_try_read ( lock_addr)
complex_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word to lock.

Description

The **lock_read** kernel service locks the specified complex lock in shared-read mode; it blocks if the lock is locked in exclusive-write mode. The lock must previously have been initialized with the **lock_init** kernel service. The **lock_read** kernel service has no return values.

The **lock_try_read** kernel service tries to lock the specified complex lock in shared-read mode; it returns immediately if the lock is locked in exclusive-write mode, otherwise it locks the lock in shared-read mode. The lock must previously have been initialized with the **lock_init** kernel service.

Execution Environment

The **lock_read** and **lock_try_read** kernel services can be called from the process environment only.

Return Values

The **lock_try_read** kernel service has the following return values:

Item	Description
TRUE	Indicates that the lock was successfully acquired in shared-read mode.
FALSE	Indicates that the lock was not acquired.

Related reference:

"lock_init Kernel Service" on page 341

"lock_islocked Kernel Service" on page 341

"lock_done Kernel Service" on page 339

Related information:

Understanding Locking

Locking Kernel Services

lock_read_to_write or lock_try_read_to_write Kernel Service

Purpose

Upgrades a complex lock from shared-read mode to exclusive-write mode.

Syntax

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

boolean_t lock_read_to_write ( lock_addr)
complex_lock_t lock_addr;

boolean_t lock_try_read_to_write ( lock_addr)
complex_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word to be converted from read-shared to write-exclusive mode.

Description

The **lock_read_to_write** and **lock_try_read_to_write** kernel services try to upgrade the specified complex lock from shared-read mode to exclusive-write mode. The lock is successfully upgraded if no other thread has already requested write-exclusive access for this lock. If the lock cannot be upgraded, it is no longer held on return from the **lock_read_to_write** kernel service; it is still held in shared-read mode on return from the **lock_try_read_to_write** kernel service.

The calling kernel thread must hold the lock in shared-read mode.

Execution Environment

The **lock_read_to_write** and **lock_try_read_to_write** kernel services can be called from the process environment only.

Return Values

The following only apply to **lock_read_to_write**:

Item	Description
TRUE	Indicates that the lock was not upgraded and is no longer held.
FALSE	Indicates that the lock was successfully upgraded to exclusive-write mode.

The following only apply to **lock_try_read_to_write**:

Item	Description
TRUE	Indicates that the lock was successfully upgraded to exclusive-write mode.
FALSE	Indicates that the lock was not upgraded and is held in read mode.

Related reference:

"lock_init Kernel Service" on page 341

"lock_islocked Kernel Service" on page 341

"lock_done Kernel Service" on page 339

Related information:

Understanding Locking

Locking Kernel Services

lock_set_recursive Kernel Service

Purpose

Prepares a complex lock for recursive use.

Syntax

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

void lock_set_recursive ( lock_addr)
complex_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word to be prepared for recursive use.

Description

The **lock_set_recursive** kernel service prepares the specified complex lock for recursive use. A complex lock cannot be nested until the **lock_set_recursive** kernel service is called for it. The calling kernel thread must hold the specified complex lock in write-exclusive mode.

When a complex lock is used recursively, the **lock_done** kernel service must be called once for each time that the thread is locked in order to unlock the lock.

Only the kernel thread which calls the **lock_set_recursive** kernel service for a lock may acquire that lock recursively.

Execution Environment

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

Return Values

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

Related reference:
"lock_init Kernel Service" on page 341
"lock_done Kernel Service" on page 339
"lock_write or lock_try_write Kernel Service"
"lock_clear_recursive Kernel Service" on page 338
Related information:
Understanding Locking
Locking Kernel Services

lock_write or lock_try_write Kernel Service

Purpose

Locks a complex lock in exclusive-write mode.

Syntax

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

```
void lock_write ( lock_addr)
complex_lock_t lock_addr;

boolean_t lock_try_write ( lock_addr)
complex_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word to lock.

Description

The **lock_write** kernel service locks the specified complex lock in exclusive-write mode; it blocks if the lock is busy. The lock must have been previously initialized with the **lock_init** kernel service. The **lock_write** kernel service has no return values.

The **lock_try_write** kernel service tries to lock the specified complex lock in exclusive-write mode; it returns immediately without blocking if the lock is busy. The lock must have been previously initialized with the **lock_init** kernel service.

Execution Environment

The **lock_write** and **lock_try_write** kernel services can be called from the process environment only.

Return Values

The **lock_try_write** kernel service has the following parameters:

Item	Description
TRUE	Indicates that the lock was successfully acquired.
FALSE	Indicates that the lock was not acquired.

Related reference:

"lock_done Kernel Service" on page 339

"lock_read_to_write or lock_try_read_to_write Kernel Service" on page 345

Related information:

Understanding Locking

Locking Kernel Services

lock_write_to_read Kernel Service

Purpose

Downgrades a complex lock from exclusive-write mode to shared-read mode.

Syntax

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

void lock_write_to_read ( lock_addr)
complex_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word to be downgraded from exclusive-write to shared-read mode.

Description

The **lock_write_to_read** kernel service downgrades the specified complex lock from exclusive-write mode to shared-read mode. The calling kernel thread must hold the lock in exclusive-write mode.

Once the lock has been downgraded to shared-read mode, other kernel threads will also be able to acquire it in shared-read mode.

Execution Environment

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

Return Values

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

Related reference:

"lock_islocked Kernel Service" on page 341

"lock_read_to_write or lock_try_read_to_write Kernel Service" on page 345

Related information:

Understanding Locking

loifp Kernel Service

Purpose

Returns the address of the software loopback interface structure.

Syntax

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

struct ifnet *loifp ()
```

Description

The **loifp** kernel service returns the address of the **ifnet** structure associated with the software loopback interface. The interface address can be used to examine the interface flags. This address can also be used to determine whether the **looutput** kernel service can be called to send a packet through the loopback interface.

Execution Environment

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

Return Values

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

Related reference:

"looutput Kernel Service" on page 352

Related information:

Network Kernel Services

longjmpx Kernel Service

Purpose

Allows exception handling by causing execution to resume at the most recently saved context.

Syntax

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

int longjmpx ( ret_val)
int ret_val;
```

Parameters

Item	Description
ret_val	Specifies the return value to be supplied on the return from the **setjmpx** kernel service for the resumed context. This value normally indicates the type of exception that has occurred.

Description

The **longjmpx** kernel service causes the normal execution flow to be modified so that execution resumes at the most recently saved context. The kernel mode lock is reacquired if it is necessary. The interrupt priority level is reset to that of the saved context.

The **longjmpx** service internally calls the **clrjmpx** service to remove the jump buffer specified by the *jump_buffer* parameter from the list of contexts to be resumed. The **longjmpx** service always returns a nonzero value when returning to the restored context. Therefore, if the value of the *ret_val* parameter is 0, the **longjmpx** service returns an **EINTR** value to the restored context.

If there is no saved context to resume, the system crashes.

Execution Environment

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

Return Values

A successful call to the **longjmpx** service does not return to the caller. Instead, it causes execution to resume at the return from a previous **setjmpx** call with the return value of the *ret_val* parameter.

Related reference:

"clrjmpx Kernel Service" on page 43

"setjmpx Kernel Service" on page 467

Related information:

Understanding Exception Handling

Process and Exception Management Kernel Services

lookupvp, lookupname, lookupname_cur Kernel Services

Purpose

Retrieves the v-node that corresponds to the named path.

Syntax

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

int lookupvp ( namep, flags, compvpp, crp)
char *namep;
int flags;
struct vnode **compvpp;
struct ucred *crp;

int lookupname ( namep, seg, flags, dirvpp, compvpp, crp)
char *namep;
int seg;
int flags;
struct vnode **dirvpp;
struct vnode **compvpp;
struct cred *crp;

int lookupname_cur ( namep, seg, flags, dirvpp, compvpp, curdvp, crp)
char *namep;
int seg;
int flags;
struct vnode **dirvpp;
struct vnode **compvpp;
struct vnode **curdvp;
struct cred *crp;
```

Parameters

Item **Description**

crp Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

namep Points to a character string path name.

flags Specifies lookup directives, including these six flags:

L_LOC The path-name resolution must not cross a mount point into another file system implementation.

L_NOFOLLOW
If the final component of the path name resolves to a symbolic link, the link is not to be traversed.

L_NOXMOUNT
If the final component of the path name resolves to a mounted-over object, the mounted-over object, rather than the root of the next virtual file system, is to be returned.

L_CRT The object is to be created.

L_DEL The object is to be deleted.

L_EROFS
An error is to be returned if the object resides in a read-only file system.

seg Specifies whether the *namep* buffer is in user space (UIO_USERSPACE) or kernel space (UIO_SYSSPACE).

compvpp Points to the location where the vnode pointer for the named object is to be returned to the calling routine.

dirvpp Points to the location where the vnode pointer for the directory containing the named object is to be returned.

curdvp Points to the vnode for a current directory to be used instead of u_cdir.

Description

The **lookupvp** kernel service provides translation of the path name provided by the *namep* parameter into a virtual file system node. The **lookupvp** service provides a flexible interface to path-name resolution by regarding the *flags* parameter values as directives to the lookup process. The lookup process is a cooperative effort between the logical file system and underlying virtual file systems (VFS). Several v-node and VFS operations are employed to:

- Look up individual name components
- Read symbolic links
- Cross mount points

The **lookupvp** kernel service determines the process's current and root directories by consulting the `u_cdir` and `u_rdir` fields in the **u** structure. Information about the virtual file system and file system installation for transient v-nodes is obtained from each name component's **vfs** or **gfs** structure. The **lookupvp** kernel service assumes that the named path is in kernel address space.

The **lookupname** kernel service provides the same service as the **lookupvp** kernel service, but allows the caller to specify whether the path name is in kernel or user space. It also provides the ability to retrieve the vnode for the directory containing the named object. The **lookupname_cur** kernel service further extends the interface by allowing the lookup to proceed relative to the given *curdvp* directory.

The vnodes returned by the **lookup** services are held. The calling routine is responsible for releasing the hold by calling the **vnop_rele** entry point when it completes its operation.

Execution Environment

The **lookup** kernel services can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
errno	Indicates an error. This number is defined in the **/usr/include/sys/errno.h** file.

Related information:

Understanding Data Structures and Header Files for Virtual File Systems

Virtual File System Overview

Virtual File System (VFS) Kernel Services

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

Item	Description
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

Item	Description
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.

Related reference:

"loifp Kernel Service" on page 349

Related information:

Network Kernel Services

ltpin Kernel Service

Purpose

Pins the address range in the system (kernel) space and frees the page space for the associated pages.

Syntax

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

int ltpin (addr, length)
caddr_t  addr;
int      length;
```

Parameters

Item	Description
addr	Specifies the address of the first byte to pin.
length	Specifies the number of bytes to pin.

Description

The **ltpin** (long term pin) kernel service pins the real memory pages touched by the address range specified by the *addr* and *length* parameters in the system (kernel) address space. It pins the real-memory pages to ensure that page faults do not occur for memory references in this address range. The **ltpin** kernel service increments the long-term pin count for each real-memory page. While either the long-term or short-term pin count is nonzero, the page cannot be paged out of real memory.

The **ltpin** kernel service pins either the entire address range or none of it. Only a limited number of pages are pinned in the system. If there are not enough unpinned pages in the system, the **ltpin** kernel service returns an error code. The **ltpin** kernel service is not a published interface.

Note: The operating system pins only whole pages at a time. Therfore, if the requested range is not aligned on a page boundary, then memory outside this range is also pinned.

The **ltpin** kernel service can only be called for addresses within the system (kernel) address space.

Return Values

Item	Description
0	Indicates successful completion.
EINVAL	Indicates that the *length* parameter has a negative value. Otherwise, the area of memory beginning at the address of the first byte to pin (the **addr** parameter) and extending for the number of bytes specified by the *length* parameter is not defined.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOMEM	Indicates that the **pin** kernel service was unable to pin due to insufficient real memory or exceeding the system-wide pin count.
ENOSPC	Indicates insufficient file system or paging space.

Related reference:

"ltunpin Kernel Service"

ltunpin Kernel Service

Purpose

Unpins the address range in system (kernel) address space and reallocates paging space for the specified region.

Syntax

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

int   ltunpin (addr, length)
caddr_t  addr;
int      length;
```

Parameters

Item	Description
addr	Specifies the address of the first byte to unpin.
length	Specifies the number of bytes to unpin.

Description

The **ltunpin** kernel service decreases the long-term pin count of each page in the address range. When the long-term pin count becomes 0, the backing storage (paging space) for the memory region is allocated and assigned to the pages. When both the long-term and short-term pin counts are 0, the page is no longer pinned and the **ltunpin** kernel service will assert. If allocating backing pages would put the system below the low paging space threshold, the call waits until paging space becomes available.

The **ltunpin** kernel service can only be called with addresses in the system (kernel) address space from the process environment.

Return Values

Item	Description
0	Indicates successful completion.
EINVAL	Indicates that the *length* parameter is a negative value.
EIO	Indicates that a permanent I/O error occurred while referencing data.

Related reference:

"ltpin Kernel Service" on page 353

m

The following kernel services begin with the with the letter m.

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

Item	Description
m	Specifies the **mbuf** chain to be adjusted.
diff	Specifies the number of bytes to be removed.

Description

The **m_adj** kernel service adjusts the size of an **mbuf** chain by the number of bytes specified by the *diff* parameter. If the number specified by the *diff* parameter is nonnegative, the bytes are removed from the front of the chain. If this number is negative, the alteration is done from back to front.

Execution Environment

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

Return Values

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

Related information:

I/O Kernel Services

mbreq Structure for mbuf Kernel Services

Purpose

Contains **mbuf** structure registration information for the **m_reg** and **m_dereg** kernel services.

Syntax

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

struct mbreq {
                int low_mbuf;
                int low_clust;
                int initial_mbuf;
                int initial_clust;
}
```

Parameters

Item	Description
low_mbuf	Specifies the **mbuf** structure low-water mark.
low_clust	Specifies the page-sized **mbuf** structure low-water mark.
initial_mbuf	Specifies the initial allocation of **mbuf** structures.
initial_clust	Specifies the initial allocation of page-sized **mbuf** structures.

Description

The **mbreq** structure specifies the **mbuf** structure usage expectations for a user of **mbuf** kernel services.

Related reference:

"m_dereg Kernel Service" on page 363

"m_reg Kernel Service" on page 372

Related information:

I/O Kernel Services

mbstat Structure for mbuf Kernel Services

Purpose

Contains **mbuf** usage statistics.

Syntax

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

struct mbstat {
ulong m_mbufs;
ulong m_clusters;
ulong m_spare;
ulong m_clfree;
ulong m_drops;
ulong m_wait;
ulong m_drain;
short m_mtypes[256];
}
```

Parameters

Item	Description
m_mbufs	Specifies the number of **mbuf** structures allocated.
m_clusters	Specifies the number of clusters allocated.
m_spare	Specifies the spare field.
m_clfree	Specifies the number of free clusters.
m_drops	Specifies the times failed to find space.
m_wait	Specifies the times waited for space.
m_drain	Specifies the times drained protocols for space.
m_mtypes	Specifies the type-specific **mbuf** structure allocations.

Description

The **mbstat** structure provides usage information for the **mbuf** services. Statistics can be viewed through the **netstat -m** command.

Related information:

netstat subroutine

I/O Kernel Services

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

Item	Description
m	Specifies the **mbuf** chain to be appended to.
n	Specifies the **mbuf** chain to append.

Description

The **m_cat** kernel service appends an **mbuf** chain specified by the *n* parameter to the end of **mbuf** chain specified by the *m* parameter. Where possible, compaction is performed.

Execution Environment

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

Return Values

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

Related information:

I/O Kernel Services

m_clattach Kernel Service

Purpose

Allocates an **mbuf** structure and attaches an external cluster.

Syntax

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

struct mbuf *
m_clattach( ext_buf,  ext_free,  ext_size,  ext_arg,  wait)
caddr_t ext_buf;
int (*ext_free)();
int ext_size;
int ext_arg;
int wait;
```

Parameters

Item	Description
ext_buf	Specifies the address of the external data area.
ext_free	Specifies the address of a function to be called when this **mbuf** structure is freed.
ext_size	Specifies the length of the external data area.
ext_arg	Specifies an argument to pass to the above function.
wait	Specifies either the **M_WAIT** or **M_DONTWAIT** value.

Description

The **m_clattach** kernel service allocates an **mbuf** structure and attaches the cluster specified by the *ext_buf* parameter. This data is owned by the caller. The `m_data` field of the returned **mbuf** structure points to the caller's data. Interrupt handlers can call this service only with the *wait* parameter set to **M_DONTWAIT**.

Note: The **m_clattach** kernel service replaces the **m_clgetx** kernel service, which is no longer supported.

The calling function is required to fill out the mbuf structure sufficiently to support normal usage. This includes support for the DMA functions during network transmission. To support DMA functions, the **ext_hasxm** flag field needs to be set to true and the **ext_xmemd** structure needs to be filled out. For buffers allocated from the kernel pinned heap, the **ext_xmemd.aspace_id** field should be set to XMEM_GLOBAL.

Execution Environment

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

Return Values

The **m_clattach** kernel service returns the address of an allocated **mbuf** structure. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structures, the **m_clattach** service returns null.

Related information:

I/O Kernel Services

m_clget Macro for mbuf Kernel Services

Purpose

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

Syntax

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

int m_clget ( m)
struct mbuf *m;
```

Parameter

Item	Description
m	Specifies the **mbuf** structure with which the cluster is to be associated.

Description

The **m_clget** macro allocates a page-sized **mbuf** cluster and attaches it to the given **mbuf** structure. If successful, the length of the **mbuf** structure is set to **CLBYTES**.

Execution Environment

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

Return Values

Item	Description
1	Indicates successful completion.
0	Indicates an error.

Related reference:

"m_clgetm Kernel Service"

Related information:

I/O Kernel Services

m_clgetm Kernel Service

Purpose

Allocates and attaches an external buffer.

Syntax

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

int
m_clgetm( m,  how,  size)
struct mbuf *m;
int  how;
int  size;
```

Parameters

Item	Description
m	Specifies the **mbuf** structure that the cluster will be associated with.
how	Specifies either the **M_DONTWAIT** or **M_WAIT** value.
size	Specifies the size of external cluster to attach. Any value less than **MAXALLOCSAVE** is valid. For larger values, **M_WAIT** must be specified.

Description

The **m_clgetm** service allocates an **mbuf** cluster of the specified number of bytes and attaches it to the **mbuf** structure indicated by the *m* parameter. If successful, the **m_clgetm** service sets the **M_EXT** flag.

Execution Environment

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

An interrupt handler can specify the *wait* parameter as **M_DONTWAIT** only.

Return Values

Item	Description
1	Indicates a successful operation.

If there are no free **mbuf** structures, the **m_clgetm** kernel service returns a null value.

Related reference:

"m_freem Kernel Service" on page 365

"m_get Kernel Service" on page 366

"m_clget Macro for mbuf Kernel Services" on page 359

Related information:

I/O Kernel Services

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

Item	Description
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.

Related information:

I/O Kernel Services

m_copy Macro for mbuf Kernel Services

Purpose

Creates a copy of all or part of a list of **mbuf** structures.

Syntax

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

struct mbuf *m_copy ( m,  off,  len)
struct mbuf *m;
int off;
int len;
```

Parameters

Item	Description
m	Specifies the **mbuf** structure, or the head of a list of **mbuf** structures, to be copied.
off	Specifies an offset into data from which copying starts.
len	Denotes the total number of bytes to copy.

Description

The **m_copy** macro makes a copy of the structure specified by the *m* parameter. The copy begins at the specified bytes (represented by the *off* parameter) and continues for the number of bytes specified by the *len* parameter. If the *len* parameter is set to **M_COPYALL**, the entire **mbuf** chain is copied.

Execution Environment

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

Return Values

Upon successful completion, the address of the copied list (the **mbuf** structure that heads the list) is returned. If the copy fails, a value of null is returned.

Related reference:

"m_copydata Kernel Service"

"m_copym Kernel Service" on page 363

Related information:

I/O Kernel Services

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

Item	Description
m	Indicates the **mbuf** structure, or the head of a list of **mbuf** structures, to be copied.
off	Specifies an offset into data from which copying starts.
len	Denotes the total number of bytes to copy.
cp	Points to a data buffer into which to copy the **mbuf** data.

Description

The **m_copydata** kernel service makes a copy of the structure specified by the *m* parameter. The copy begins at the specified bytes (represented by the *off* parameter) and continues for the number of bytes specified by the *len* parameter. The data is copied into the buffer specified by the *cp* parameter.

Execution Environment

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

Return Values

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

Related reference:

"m_copy Macro for mbuf Kernel Services" on page 361

Related information:

I/O Kernel Services

m_copym Kernel Service

Purpose

Creates a copy of all or part of a list of **mbuf** structures.

Syntax

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

struct mbuf *
m_copym( m,  off,  len,  wait)
struct mbuf  m;
int  off;
int  len;
int  wait;
```

Parameters

Item	Description
m	Specifies the **mbuf** structure to be copied.
off	Specifies an offset into data from which copying will start.
len	Specifies the total number of bytes to copy.
wait	Specifies either the **M_DONTWAIT** or **M_WAIT** value.

Description

The **m_copym** kernel service makes a copy of the **mbuf** structure specified by the *m* parameter starting at the specified offset from the beginning and continuing for the number of bytes specified by the *len* parameter. If the *len* parameter is set to **M_COPYALL**, the entire **mbuf** chain is copied.

If the **mbuf** structure specified by the *m* parameter has an external buffer attached (that is, the **M_EXT** flag is set), the copy is done by reference to the external cluster. In this case, the data must not be altered or both copies will be changed. Interrupt handlers can specify the *wait* parameter as **M_DONTWAIT** only.

Execution Environment

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

Return Values

The address of the copy is returned upon successful completion. If the copy fails, null is returned. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structures, the **m_copym** kernel service returns a null value.

Related reference:

"m_copydata Kernel Service" on page 362

"m_copy Macro for mbuf Kernel Services" on page 361

Related information:

I/O Kernel Services

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

Item	Description
mbp	Defines the address of an **mbreq** structure that specifies expected **mbuf** usage.

Description

The **m_dereg** kernel service deregisters requirements previously registered with the **m_reg** kernel service. The **m_dereg** service is mandatory if the **m_reg** service is called.

Execution Environment

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

Return Values

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

Related reference:

"mbreq Structure for mbuf Kernel Services" on page 355

"m_reg Kernel Service" on page 372

Related information:

I/O Kernel Services

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

Item	Description
m	Specifies the **mbuf** structure to be freed.

Description

The **m_free** kernel service returns an **mbuf** structure to the buffer pool. If the **mbuf** structure specified by the *m* parameter has an attached cluster (that is, a paged-size **mbuf** structure), the **m_free** kernel service also frees the associated external storage.

Execution Environment

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

Return Values

If the **mbuf** structure specified by the *m* parameter is the head of an **mbuf** chain, the **m_free** service returns the next **mbuf** structure in the chain. A null value is returned if the structure specified by the *m* parameter is not part of an **mbuf** chain.

Related reference:

"m_get Kernel Service" on page 366

Related information:

I/O Kernel Services

m_freem Kernel Service

Purpose

Frees an entire **mbuf** chain.

Syntax

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

void m_freem ( m)
struct mbuf *m;
```

Parameter

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

Description

The **m_freem** kernel service starts the **m_free** kernel service for each **mbuf** structure in the chain headed by the head specified by the *m* parameter.

Execution Environment

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

Return Values

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

Related reference:

"m_free Kernel Service" on page 364

"m_get Kernel Service"

Related information:

I/O Kernel Services

m_get Kernel Service

Purpose

Allocates a memory buffer (mbuf) from the **mbuf** pool.

Syntax

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

struct mbuf *m_get ( wait,  type)
int  wait;
int  type;
```

Parameters

Item	Description
wait	Indicates the action to be taken if there are no free **mbuf** structures. Possible values are: **M_DONTWAIT** Called from either an interrupt or process environment. **M_WAIT** Called from a process environment.
type	Specifies a valid **mbuf** type, as listed in the **/usr/include/sys/mbuf.h** file.

Description

The **m_get** kernel service allocates an **mbuf** structure of the specified type. If the buffer pool is empty and the *wait* parameter is set to **M_WAIT**, the **m_get** kernel service does not return until an **mbuf** structure is available.

Execution Environment

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

An interrupt handler can specify the *wait* parameter as **M_DONTWAIT** only.

Return Values

Upon successful completion, the **m_get** service returns the address of an allocated **mbuf** structure. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structures, the **m_get** kernel service returns a null value.

Related reference:

"m_free Kernel Service" on page 364

"m_freem Kernel Service" on page 365

Related information:

I/O Kernel Services

m_getclr Kernel Service

Purpose

Allocates and zeroes a memory buffer from the **mbuf** pool.

Syntax

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

struct mbuf *m_getclr ( wait,  type)
int wait;
int type;
```

Parameters

Item	Description
wait	This flag indicates the action to be taken if there are no free **mbuf** structures. Possible values are: **M_DONTWAIT** Called from either an interrupt or process environment. **M_WAIT** Called from a process environment only.
type	Specifies a valid **mbuf** type, as listed in the **/usr/include/sys/mbuf.h** file.

Description

The **m_getclr** kernel service allocates an **mbuf** structure of the specified type. If the buffer pool is empty and the *wait* parameter is set to **M_WAIT** value, the **m_getclr** service does not return until an **mbuf** structure is available.

The **m_getclr** kernel service differs from the **m_get** kernel service in that the **m_getclr** service zeroes the data portion of the allocated **mbuf** structure.

Execution Environment

The **m_getclr** kernel service can be called from either the process or interrupt environment. Interrupt handlers can call the **m_getclr** service only with the *wait* parameter set to the **M_DONTWAIT** value.

Return Values

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

Related reference:

"m_free Kernel Service" on page 364

"m_freem Kernel Service" on page 365

"m_get Kernel Service" on page 366

Related information:

I/O Kernel Services

m_getclust Macro for mbuf Kernel Services

Purpose

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

Syntax

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

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

Parameters

Item	Description
wait	Indicates the action to be taken if there are no available **mbuf** structures. Possible values are: **M_DONTWAIT** Called from either an interrupt or process environment. **M_WAIT** Called from a process environment only.
type	Specifies a valid **mbuf** type from the **/usr/include/sys/mbuf.h** file.

Description

The **m_getclust** macro allocates an **mbuf** structure of the specified type. If the allocation succeeds, the **m_getclust** macro then attempts to attach a page-sized cluster to the structure.

If the buffer pool is empty and the *wait* parameter is set to **M_WAIT**, the **m_getclust** macro does not return until an **mbuf** structure is available.

Execution Environment

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

Return Values

The address of an allocated **mbuf** structure is returned on success. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structures, the **m_getclust** macro returns a null value.

Related reference:

"m_getclustm Kernel Service"

Related information:

I/O Kernel Services

m_getclustm Kernel Service

Purpose

Allocates an **mbuf** structure and attaches a cluster of the specified size, both from the **mbuf** buffer pool.

Syntax

```
#include <sys/mbuf.h>
#include <net/net_globals.h>

struct mbuf *
m_getclustm( wait,  type,  size)
int wait;
int type;
int size;
```

Parameters

Item	Description
wait	Specifies either the **M_DONTWAIT** or **M_WAIT** value.
type	Specifies a valid **mbuf** type from the **/usr/include/sys/mbuf.h** file.
size	Specifies the size of the external cluster to attach. Any value less than **MAXALLOCSAVE** is valid. For larger values, **M_WAIT** must be specified.

Description

The **m_getclustm** service allocates an **mbuf** structure of the specified type. If successful, the **m_getclustm** service then attempts to attach a cluster of the indicated size (specified by the *size* parameter) to the **mbuf** structure. If the buffer pool is empty and the *wait* parameter is set to **M_WAIT**, the **m_get** service does not return until an **mbuf** structure is available. Interrupt handlers should call this service only with the *wait* parameter set to **M_DONTWAIT**.

Execution Environment

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

An interrupt handler can specify the *wait* parameter as **M_DONTWAIT** only.

Return Values

The **m_getclustm** kernel service returns the address of an allocated **mbuf** structure on success. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structures, the **m_getclustm** kernel service returns null.

Related reference:
"m_clget Macro for mbuf Kernel Services" on page 359
"m_free Kernel Service" on page 364
"m_freem Kernel Service" on page 365
"m_get Kernel Service" on page 366
"m_getclust Macro for mbuf Kernel Services" on page 367
Related information:
I/O Kernel Services

m_gethdr Kernel Service

Purpose

Allocates a header memory buffer from the **mbuf** pool.

Syntax

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

struct mbuf *
m_gethdr ( wait,  type)
int  wait;
int  type;
```

Parameters

Item	Description
wait	Specifies either the **M_DONTWAIT** or **M_WAIT** value.
type	Specifies the valid **mbuf** type from the **/usr/include/sys/mbuf.h** file.

Description

The **m_gethdr** kernel service allocates an **mbuf** structure of the specified type. If the buffer pool is empty and the *wait* parameter is set to **M_WAIT**, the **m_gethdr** kernel service will not return until an **mbuf** structure is available. Interrupt handlers should call this kernel service only with the *wait* parameter set to **M_DONTWAIT**. The **M_PKTHDR** flag is set for the returned **mbuf** structure.

Execution Environment

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

An interrupt handler can specify the *wait* parameter as **M_DONTWAIT** only.

Return Values

The address of an allocated **mbuf** structure is returned on success. If the *wait* parameter is set to **M_DONTWAIT** and there are no free **mbuf** structure, the **m_gethdr** kernel service returns null.

Related Information

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

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

Related reference:

"m_free Kernel Service" on page 364

"m_freem Kernel Service" on page 365

Related information:

I/O Kernel Services

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

Item	Description
m	Indicates the address of the **mbuf** structure in question.

Description

The **M_HASCL** macro determines if an **mbuf** structure has an attached cluster.

Execution Environment

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

Example

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

```
struct mbuf  *m;
if (M_HASCL(m))
   printf("mbuf has attached cluster");
```

Related information:

I/O Kernel Services

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

Item	Description
m	Specifies the **mbuf** chain to be adjusted.
size	Specifies the number of bytes to be contiguous.

Description

The **m_pullup** kernel service guarantees that the **mbuf** structure at the head of a chain has in contiguous memory within its data area at least the number of data bytes specified by the *size* parameter.

Execution Environment

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

Return Values

Upon successful completion, the head structure in the altered **mbuf** chain is returned.

A value of null is returned and the original chain is deallocated under the following circumstances:

- The size of the chain is less than indicated by the *size* parameter.
- The number indicated by the *size* parameter is greater than the data portion of the head-size **mbuf** structure.

Related information:

I/O Kernel Services

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

Item	Description
mbp	Defines the address of an **mbreq** structure that specifies expected **mbuf** usage.

Description

The **m_reg** kernel service lets users of **mbuf** services specify initial requirements. The **m_reg** kernel service also allows the buffer pool low-water and deallocation marks to be adjusted based on expected usage. Its use is recommended for better control of the buffer pool.

When the number of free **mbuf** structures falls below the low-water mark, the total **mbuf** pool is expanded. When the number of free **mbuf** structures rises above the deallocation mark, the total **mbuf** pool is contracted and resources are returned to the system.

Execution Environment

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

Return Values

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

Related reference:

"mbreq Structure for mbuf Kernel Services" on page 355

"m_dereg Kernel Service" on page 363

Related information:

I/O Kernel Services

md_restart_block_read Kernel Service

Purpose

A copy of the **RESTART_BLOCK** structure in the **NVRAM** header will be placed in the caller's buffer.

Syntax

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

```
int md_restart_block_read (md)
                struct mdio *md;
```

Parameters

Item	Description
md	Specifies the address of the **mdio** structure. The **mdio** structure contains the following fields: **md_data** Pointer to the data buffer. **md_size** Number of bytes in the data buffer. **md_addr** Contains the value PMMode on return in the least significant byte.

Description

The RestartBlock which is in the **NVRAM** header will be copied to the user supplied buffer. This block is a communication vehicle for the software and the firmware.

Return Values

Returns 0 for successful completion.

Item	Description
ENOMEM	Indicates that there was not enough room in the user supplied buffer to contain the RestartBlock.
EINVAL	Indicates this is not a PowerPC® reference platform.

Prerequisite Information

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

Related information:

Machine Device Driver

md_restart_block_upd Kernel Service

Purpose

The caller supplied RestartBlock will be copied to the **NVRAM** header.

Syntax

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

int md_restart_block_upd (md, pmmode)
                struct mdio *md;
                unsigned char pmmode;
```

Description

The 8-bit value in *pmmode* will be stored into the **NVRAM** header at the PMMode offset.The RestartBlock which is in the caller's buffer will be copied to the **NVRAM** after the RestartBlock checksum is calculated and a new Crc1 value is computed.

Parameters

Item	Description
md	Specifies the address of the **mdio** structure. The **mdio** structure contains the following fields: **md_data** Pointer to the RestartBlock structure..
pmmode	Value to be stored into PMMode in the **NVRAM** header.

Return Values

Returns 0 for successful completion.

Item	Description
EINVAL	Indicates this is not a PowerPC reference platform.

Prerequisite Information

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

Related information:

Machine Device Driver

MTOCL Macro for mbuf Kernel Services

Purpose

Converts a pointer to an **mbuf** structure to a pointer to the head of an attached cluster.

Syntax

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

struct mbuf * m;
MTOCL (m);
```

Parameter

Item	Description
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.

Related reference:

“M_HASCL Macro for mbuf Kernel Services” on page 370

Related information:

I/O Kernel Services

MTOD Macro for mbuf Kernel Services

Purpose

Converts a pointer to an **mbuf** structure to a pointer to the data stored in that **mbuf** structure.

Syntax

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

MTOD ( m,  type);
```

Parameters

Item	Description
m	Identifies the address of an **mbuf** structure.
type	Indicates the type to which the resulting pointer should be cast.

Description

The **MTOD** macro converts a pointer to an **mbuf** structure into a pointer to the data stored in the **mbuf** structure. This macro can be used as in the following example:

```
char    *bufp;
         bufp = MTOD(m, char *);
```

Execution Environment

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

Related reference:

"DTOM Macro for mbuf Kernel Services" on page 105

Related information:

I/O Kernel Services

M_XMEMD Macro for mbuf Kernel Services

Purpose

Returns the address of an **mbuf** cross-memory descriptor.

Syntax

```
#include   <sys/mbuf.h>
#include   <sys/xmem.h>

struct mbuf * m;
M_XMEMD (m);
```

Parameter

Item	Description
m	Specifies the address of the **mbuf** structure in question.

Description

The **M_XMEMD** macro returns the address of an **mbuf** cross-memory descriptor.

Execution Environment

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

Example

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

```
struct mbuf   *m;
struct xmem   *xmemd;

xmemd = M_XMEMD(m);
```

Related information:

I/O Kernel Services

mycpu Kernel Service

Purpose

Gets the bind ID of the processor we are running on.

Syntax

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

cpu_t myc ()
```

Description

The **mycpu** kernel service returns the bind ID of the processor we are currently running on.

Execution Environment

The **mycpu** kernel services can be called from either the process or interrupt environment. This routine must be called disabled. Otherwise, the calling thread might be preempted and resume execution on a different processor resulting in a stale value being returned.

Return Values

The **mycpu** kernel service returns the bind ID of the current processor.

Related reference:

"bindprocessor Kernel Service" on page 28

n

The following kernel services begin with the with the letter n.

nameToXfid() Kernel Service

Purpose

Obtains the `xfid` value and attributes for a specific file name.

Syntax

```
#include <sys/xfops.h>
#include <sys/vattr.h>

int     nameToXfid(char *pathname,
                         struct xfid *xfp,
                         struct vattr *vap,
                         long flags);
```

Description

A kernel extension might need to convert a path name to an `xfid_t` structure. The `nameToXfid()` kernel service returns the `xfid` value for a specific path name.

Parameters

pathname
Full path name of the file for which an `xfid` value is needed.

xfp
Pointer to an `xfid_t` structure to hold the `xfid` value that is set by this routine.

vap
Pointer to a `vattr` structure to be entered by this routine. No attributes are set if the pointer is null.

flags
Operation modifiers. This parameter must be set to zero.

Return values

0 Indicates success. The `xfid` value and the optional `vattr` structure are returned.

ENOENT
Name not found.

EPERM
No permission for lookup.

EINVAL
Invalid parameter is specified.

net_attach Kernel Service

Purpose

Opens a communications I/O device handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <aixif/net_if.h>
#include <sys/comio.h>

int net_attach (kopen_ext, device_req, netid, netfpp)
struct  kopen_ext * kopen_ext;
struct  device_req * device_req;
struct  netid_list * netid;
struct file  ** netfpp;
```

Parameters

Item	Description
kopen_ext	Specifies the device handler kernel open extension.
device_req	Indicates the address of the device description structure.
netid	Indicates the address of the network ID list.
netfpp	Specifies the address of the variable that will hold the returned file pointer.

Description

The **net_attach** kernel service opens the device handler specified by the *device_req* parameter and then starts all the network IDs listed in the address specified by the *netid* parameter. The **net_attach** service then sleeps and waits for the asynchronous `start completion` notifications from the **net_start_done** kernel service.

Execution Environment

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

Return Values

Upon success, a value of 0 is returned and a file pointer is stored in the address specified by the *netfpp* parameter. Upon failure, the **net_attach** service returns either the error codes received from the **fp_opendev** or **fp_ioctl** kernel service, or the value **ETIMEDOUT**. The latter value is returned when an open operation times out.

Related reference:

"net_detach Kernel Service"

"net_start Kernel Service" on page 381

"net_start_done Kernel Service" on page 382

Related information:

Network Kernel Services

net_detach Kernel Service

Purpose

Closes a communications I/O device handler.

Syntax

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

int net_detach ( netfp)
struct file   *netfp;
```

Parameter

Item	Description
netfp	Points to an open file structure obtained from the **net_attach** kernel service.

Description

The **net_detach** kernel service closes the device handler associated with the file pointer specified by the *netfp* parameter.

Execution Environment

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

Return Values

The **net_detach** service returns the value it obtains from the **fp_close** service.

Related reference:

"fp_close Kernel Service" on page 144

"net_attach Kernel Service" on page 377

Related information:

Network Kernel Services

net_error Kernel Service

Purpose

Handles errors for communication network interface drivers.

Syntax

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

net_error ( ifp,  error_code,  netfp)
struct  ifnet *ifp;
int error_code;
struct  file *netfp;
```

Parameters

Item	Description
error_code	Specifies the error code listed in the **/usr/include/sys/comio.h** file.
ifp	Specifies the address of the **ifnet** structure for the device with an error.
netfp	Specifies the file pointer for the device with an error.

Description

The **net_error** kernel service provides generic error handling for communications network interface (**if**) drivers. Network interface (**if**) kernel extensions call this service to trace errors and, in some instances, perform error recovery.

Errors traced include those:

- Received from the communications adapter drivers.
- Occurring during input and output packet processing.

Execution Environment

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

Return Values

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

Related reference:

"net_attach Kernel Service" on page 377

"net_detach Kernel Service" on page 378

Related information:

Network Kernel Services

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

Item	Description
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

Item	Description
0	Indicates that the sleeping process was not awakened by a signal.
1	Indicates that the sleeper was awakened by a signal.

Related reference:

"net_wakeup Kernel Service" on page 382

"sleep Kernel Service" on page 475

Related information:

Network Kernel Services

net_start Kernel Service

Purpose

Starts network IDs on a communications I/O device handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <aixif/net_if.h>
#include <sys/comio.h>

struct  file *net_start ( netfp,  netid)
struct  file *netfp;
struct  netid_list *netid;
```

Parameters

Item	Description
netfp	Specifies the file pointer of the device handler.
netid	Specifies the address of the network ID list.

Description

The **net_start** kernel service starts all the network IDs listed in the list specified by the *netid* parameter. This service then waits for the asynchronous notification of completion of starts.

Execution Environment

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

Return Values

The **net_start** service uses the return value returned from a call to the **fp_ioctl** service requesting the **CIO_START** operation.

Item	Description
ETIMEDOUT	Indicates that the start for at least one network ID timed out waiting for start-done notifications from the device handler.

Related reference:

"fp_ioctl Kernel Service" on page 150

"net_attach Kernel Service" on page 377

"net_start_done Kernel Service" on page 382

Related information:

Network Kernel Services

net_start_done Kernel Service

Purpose

Starts the done notification handler for communications I/O device handlers.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <aixif/net_if.h>
#include <sys/comio.h>

void net_start_done ( netid,  sbp)
struct  netid_list *netid;
struct  status_block *sbp;
```

Parameters

Item	Description
netid	Specifies the address of the network ID list for the device being started.
sbp	Specifies the status block pointer returned from the device handler.

Description

The **net_start_done** kernel service is used to mark the completion of a network ID start operation. When all the network IDs listed in the *netid* parameter have been started, the **net_attach** kernel service returns to the caller. The **net_start_done** service should be called when a **CIO_START_DONE** status block is received from the device handler. If the status block indicates an error, the start process is immediately aborted.

Execution Environment

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

Return Values

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

Related reference:

"net_attach Kernel Service" on page 377

"net_start Kernel Service" on page 381

Related information:

CIO_START_DONE subroutine

Network Kernel Services

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

Item	Description
chan	Specifies the wait channel.

Description

The **net_wakeup** service wakes up all network processes sleeping on the specified wait channel.

Execution Environment

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

Return Values

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

Related reference:

"net_sleep Kernel Service" on page 380

Related information:

Network Kernel Services

net_xmit Kernel Service

Purpose

Transmits data using a communications device handler .

Syntax

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

int net_xmit (ifp, m, netfp, lngth, m_ext)
struct  ifnet * ifp;
struct  mbuf * m;
struct  file * netfp;
int   lngth;
struct mbuf * m_ext;
```

Parameters

Item	Description
ifp	Indicates an address of the **ifnet** structure for this interface.
m	Specifies the address of an **mbuf** structure containing the data to transmit.
netfp	Indicates the open file pointer obtained from the **net_attach** kernel service.
lngth	Indicates the total length of the buffer being transmitted.
m_ext	Indicates the address of an **mbuf** structure containing a write extension.

Description

The **net_xmit** kernel service builds a **uio** structure and then invokes the **fp_rwuio** service to transmit a packet. The **net_xmit_trace** kernel service is an alternative for network interfaces that choose not to use the **net_xmit** kernel service.

Execution Environment

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

Return Values

Item	Description
0	Indicates that the packet was transmitted successfully.
ENOBUFS	Indicates that buffer resources were not available.

The **net_xmit** kernel service returns a value from the **fp_rwuio** service when an error occurs during a call to that service.

Related reference:

"fp_rwuio Kernel Service" on page 165

"net_xmit_trace Kernel Service"

Related information:

Network Kernel Services

net_xmit_trace Kernel Service

Purpose

Traces transmit packets.

Syntax

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

int net_xmit_trace (  ifp,  mbuf )
struct ifnet *ifp;
struct mbuf *mbuf;
```

Parameters

Item	Description
ifp	Designates the address of the **ifnet** structure for this interface.
mbuf	Designates the address of the **mbuf** structure to be traced.

Description

The **net_xmit_trace** kernel service traces the data pointed to by the *mbuf* parameter. This kernel service was added for those network interfaces that choose not to use the **net_xmit** kernel service to transmit packets. An application program (the **iptrace** command) reads the trace data and writes it to a file for the **ipreport** command to interpret.

Execution Environment

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

Return Values

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

Related reference:

"net_xmit Kernel Service" on page 383

Related information:
ipreport subroutine
iptrace subroutine
Network Kernel Services

NLuprintf Kernel Service

Purpose

Submits a request to print an internationalized message to a process' controlling terminal.

Syntax

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

int NLuprintf (Uprintf)
struct uprintf *Uprintf;
```

Parameters

Item	Description
Uprintf	Points to a **uprintf** request structure.

Description

The **NLuprintf** kernel service submits a internationalized kernel message request with the **uprintf** request structure specified by the *Uprintf* parameter as input. Once the request has been successfully submitted, the **uprintfd** daemon retrieves, converts, formats, and writes the message described by the **uprintf** request structure to a process' controlling terminal.

The caller must initialize the **uprintf** request structure before calling the **NLuprintf** kernel service. Fields in the **uprintf** request structure use several constants. The following constants are defined in the **/usr/include/sys/uprintf.h** file:

- **UP_MAXSTR**
- **UP_MAXARGS**
- **UP_MAXCAT**
- **UP_MAXMSG**

The **uprintf** request structure consists of the following fields:

Field **Description**

`Uprintf->upf_defmsg`

Points to a default message format. The default message format is a character string that contains either or both of two types of objects:

- Plain characters, which are copied to the message output stream
- Conversion specifications, each of which causes zero or more items to be fetched from the *Uprintf->arg* value parameter array

Each conversion specification consists of a % (percent sign) followed by a character that indicates the type of conversion to be applied:

% Performs no conversion. Prints a % character.

d, i Accepts an integer value and converts it to signed decimal notation.

u Accepts an integer value and converts it to unsigned decimal notation.

o Accepts an integer value and converts it to unsigned octal notation.

x Accepts an integer value and converts it to unsigned hexadecimal notation.

c Accepts and prints a **char** value.

s Accepts a value as a string (character pointer). Characters from the string are printed until a \0 (null character) is encountered.

Field-width or precision conversion specifications are not supported.

The maximum length of the default message-format string pointed to by the `Uprintf->upf_defmsg` field is the number of characters specified by the **UP_MAXSTR** constant. The `Uprintf->upf_defmsg` field must be a nonnull character.

The default message format is used in constructing the kernel message if the message format described by the `Uprintf->upf_NLsetno` and `Uprint->upf_NLmsgno` fields cannot be retrieved from the message catalog specified by `Uprintf->upf_NLcatname`. The conversion specifications contained within the default message format should match those contained in the message format specified by the `upf_NLsetno` and `upf_NLmsgno` fields.

`Uprintf->upf_arg[UP_MAXARGS]`

Specifies from zero to the number of value parameters specified by the **UP_MAXARGS** constant. A *Value* parameter may be a integer value, a character value, or a string value (character pointer). Strings are limited in length to the number of characters specified by the **UP_MAXSTR** constant. String value parameters must be nonnull characters. The number, type, and order of items in the *Value* parameter array should match the conversion specifications within the message format string.

`Uprintf->upf_NLcatname`

Points to the message catalog file name. If the catalog file name referred to by the `Uprintf->upf_NLcatname` field begins with a / (slash), it is assumed to be an absolute path name. If the catalog file name is not an absolute path name, the process environment determines the directory paths to search. The maximum length of the catalog file name is limited to the number of characters specified by the **UP_MAXCAT** constant. The value of the `Uprintf->upf_NLcatname` field must be a nonnull character.

`Uprintf->upf_NLsetno`

Specifies the set ID.

Field	Description
`Uprintf->upf_NLmsgno`	Specifies the message ID. The `Uprintf->upf_NLsetno` and `Uprintf->upf_NLmsgno` fields specify a particular message format string to be retrieved from the message catalog specified by the `Uprintf->upf_NLcatname` field. The maximum length of the constructed kernel message is limited to the number of characters specified by the **UP_MAXMSG** constant. Messages larger then the number of characters specified by the **UP_MAXMSG** constant are discarded.

Execution Environment

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

Return Values

Item	Description
0	Indicates a successful operation.
ENOMEM	Indicates that memory is not available to buffer the request.
ENODEV	Indicates that a controlling terminal does not exist for the process.
ESRCH	Indicates the **uprintfd** daemon is not active. No requests may be submitted.
EINVAL	Indicates that the message catalog file-name pointer is null or the catalog file name is greater than the number of characters specified by the **UP_MAXCAT** constant.
EINVAL	Indicates that a string-value parameter pointer is null or the string-value parameter is greater than the number of characters specified by the **UP_MAXCAT** constant.
EINVAL	Indicates one of the following: • Default message format pointer is null. • Number of characters in the default message format is greater than the number specified by the **UP_MAXSTR** constant. • Number of conversion specifications contained within the default message format is greater than the number specified by the **UP_MAXARGS** constant.

Related reference:

"uprintf Kernel Service" on page 527

Related information:

uprintfd subroutine

Process and Exception Management Kernel Services

ns_add_demux Network Kernel Service

Purpose

Adds a demuxer for the specified type of network interface.

Syntax

```
#include <sys/ndd.h>
#include <sys/cdli.h>

int ns_add_demux (ndd_type, demux)
       u_long  ndd_type;
       struct ns_demuxer * demux;
```

Parameters

Item	Description
ndd_type	Specifies the interface type of the demuxer to be added.
demux	Specifies the pointer to an **ns_demux** structure that defines the demuxer.

Description

The **ns_add_demux** network service adds the specified demuxer to the list of available network demuxers. Only one demuxer per network interface type can exist. An interface type describes a certain class of network devices that have the same characteristics (such as ethernet or token ring). The values of the *ndd_type* parameter listed in the **/usr/include/sys/ndd.h** file are the numbers defined by Simple Network Management Protocol (SNMP). If the desired type is not in the **ndd.h** file, the SNMP value should be used if it is defined. Otherwise, any undefined type above **NDD_MAX_TYPE** may be used.

Note: The **ns_demuxer** structure must be allocated and pinned by the network demuxer.

Examples

The following example illustrates the **ns_add_demux** network service:

```
struct ns_demuxer demuxer;
bzero (&demuxer, sizeof (demuxer));
demuxer.nd_add_filter = eth_add_filter;
demuxer.nd_del_filter = eth_del_filter;
demuxer.nd_add_status = eth_add_status;
demuxer.nd_del_status = eth_del_status;
demuxer.nd_receive = eth_receive;
demuxer.nd_status = eth_status;
demuxer.nd_response = eth_response;
demuxer.nd_use_nsdnx = 1;
ns_add_demux(NDD_ISO88023, &demuxer);
```

Return Values

Item	Description
0	Indicates the operation was successful.
EEXIST	Indicates a demuxer already exists for the given type.

Related reference:
"ns_del_demux Network Service" on page 394

ns_add_filter Network Service

Purpose

Registers a receive filter to enable the reception of packets.

Syntax

```
#include <sys/cdli.h>
#include <sys/ndd.h>

int ns_add_filter (nddp, filter, len, ns_user)
       struct ndd * nddp;
       caddr_t  filter;
       int  len;
       struct ns_user * ns_user;
```

Parameters

Item	Description
nddp	Specifies the **ndd** structure to which this add request applies.
filter	Specifies the pointer to the receive filter.
len	Specifies the length in bytes of the receive filter to which the *filter* parameter points.
ns_user	Specifies the pointer to a **ns_user** structure that defines the user.

Description

The **ns_add_filter** network service registers a receive filter for the reception of packets and enables a network demuxer to route packets to the appropriate users. The **add** request is passed on to the **nd_add_filter** function of the demuxer for the specified NDD. The caller of the **ns_add_filter** network service is responsible for relinquishing filters before calling the **ns_free** network service.

Examples

The following example illustrates the **ns_add_filter** network service:

```
struct ns_8022 dl;
struct ns_user ns_user;

dl.filtertype = NS_LLC_DSAP_SNAP;
dl.dsap = 0xaa;
dl.orgcode[0] = 0x0;
dl.orgcode[1] = 0x0;
dl.orgcode[2] = 0x0;
dl.ethertype = 0x0800;

ns_user.isr = NULL;
ns_user.isr_data = NULL;
ns_user.protoq = &ipintrq;
ns_user.netisr = NETISR_IP;
ns_user.ifp = ifp;
ns_user.pkt_format = NS_PROTO_SNAP;

ns_add_filter(nddp, &dl, sizeof(dl), &ns_user);
```

There are two ways a user (that is, the entity that is interested in receiving incoming packets) can be invoked when a packet arrives. In the first method, a protocol queue can be defined in which incoming packets are queued upon receipt, and the specified *netisr* is scheduled to let the user know that there are new packets in the queue. For example, the preceding code assumes a network interrupt service request (netisr) with the name **NETISR_IP** has been defined. When a packet arrives for the specified user, the packet is queued on the specified protocol queue (in this case, **ipintrq**) and the **NETISR_IP** request is scheduled to be executed. Because of its complexity, this mode is not currently being used by any network user.

The preferred way of receiving incoming packets is by registering an interrupt service request (isr) function that handles incoming packets; **ns_user.isr** points to the function that will get invoked whenever a packet that matches the specified filter arrives. This function should expect the following four arguments:

```
void isr (ndd_t *nddp, mbuf *m, caddr_t macp, caddr_t extp)
```

where

Item	Description
nddp	Pointer to the **ndd** structure representing the adapter where the packet was received.
m	Pointer to the **mbuf** structure representing the packet that was received.
macp	Pointer to the start of the MAC header of the packet that was received.
extp	Pointer to the (optional) structure specified in **ns_user.isr_data**, or NULL if none was specified.

In the following code, the function **bpf_cdli_tap** will be called when a new packet arrives; a pointer to the **bp** structure will be passed as the fourth parameter when **bpf_cdli_tap** is called.

```
dl.filtertype = NS_TAP;

ns_user.isr = bpf_cdli_tap;
ns_user.isr_data = (caddr_t) bp;
ns_user.protoq = (struct ifqueue *) NULL;
ns_user.netisr = 0;
ns_user.ifp = (struct ifnet *) NULL;
ns_user.pkt_format = NS_INCLUDE_MAC;
```

Note: Both modes of receiving packets are mutually exclusive. In other words, if the **ns_user.protoq** member is non-null, the protocol queue method is used; otherwise, the direct isr function method is used, and the **ns_user.isr** function pointer must be a valid function pointer.

In both cases, **ns_user.ifp** can optionally point to the **ifnet** structure of the interface where the packets will be received. If it is non-null, the state of the interface will be verified when a packet is received. If the interface is not up, the packet will be dropped and it will not be delivered to the user. If the interface is up, the statistics for the number of received packets will be incremented, and the ifp will be saved in the packet's **mbuf** structure's **m_pkthdr.rcvif** field.

The **ns_user.pkt_format** member determines how much of the MAC header the user is interested in receiving. Its possible values are:

Item	Description
NS_PROTO	Do not include the LLC header (but include the SNAP header, if there is one).
NS_PROTO_SNAP	Do not include the LLC SNAP header (that is, remove the entire MAC header and deliver only the data).
NS_INCLUDE_LLC	Include the LLC header.
NS_INCLUDE_MAC	Include the entire MAC header.

Item | **Description**

NS_HANDLE_HEADERS
Instead of passing the specified **ns_user.isr_data** structure by itself, build an **isr_data_ext** structure containing header information, as well as a pointer to the specified **ns_user.isr_data**. These are the fields that will be set in the **isr_data_ext** structure:

isr_data_ext.isr_data
Pointer to the structure passed as **ns_user.isr_data**.

isr_data_ext.dstp
Pointer to the destination MAC address.

isr_data_ext.dstlen
Length of the destination MAC address.

isr_data_ext.srcp
Pointer to the source MAC address.

isr_data_ext.seclen
Length of the source MAC address.

isr_data_ext.segp
Pointer to the routing segment.

isr_data_ext.seglen
Length of the routing segment.

isr_data_ext.llcp
Pointer to the LLC.

isr_data_ext.llclen
Length of the LLC.

It is possible to combine **NS_HANDLE_HEADERS** with one of the other flags by means of a logical OR operator (for example, `ns_user.pkt_format = NS_INCLUDE_MAC | NS_HANDLE_HEADERS`). The other flags, however, are mutually exclusive.

Return Values

Item	Description
0	Indicates the operation was successful.

The network demuxer may supply other return values.

Related reference:

"ns_del_filter Network Service" on page 395

ns_add_status Network Service

Purpose

Adds a status filter for the routing of asynchronous status.

Syntax

```
#include <sys/cdli.h>
#include <sys/ndd.h>

int ns_add_status (nddp, statfilter, len, ns_statuser)
       struct ndd * nddp;
       caddr_t  statfilter;
       int  len;
       struct ns_statuser * ns_statuser;
```

Parameters

Item	Description
nddp	Specifies a pointer to the **ndd** structure to which this add request applies.
statfilter	Specifies a pointer to the status filter.
len	Specifies the length, in bytes, of the value of the *statfilter* parameter.
ns_statuser	Specifies a pointer to an **ns_statuser** structure that defines this user.

Description

The **ns_add_status** network service registers a status filter. The add request is passed on to the **nd_add_status** function of the demuxer for the specified network device driver (NDD). This network service enables the user to receive asynchronous status information from the specified device.

Note: The user's status processing function is specified by the `isr` field of the **ns_statuser** structure. The network demuxer calls the user's status processing function directly when asynchronous status information becomes available. Consequently; the status processing function cannot be a scheduled routine. The caller of the **ns_add_status** network service is responsible for relinquishing status filters before calling the **ns_free** network service.

Examples

The following example illustrates the **ns_add_status** network service:

```
struct ns_statuser   user;
struct ns_com_status   filter;

filter.filtertype = NS_STATUS_MASK;
filter.mask = NDD_HARD_FAIL;
filter.sid = 0;
user.isr = status_fn;
user.isr_data = whatever_makes_sense;

error = ns_add_status(nddp, &filter, sizeof(filter), &user);
```

Return Values

Item	Description
0	Indicates the operation was successful.

The network demuxer may supply other return values.

Related reference:

"ns_del_status Network Service" on page 396

ns_alloc Network Service

Purpose

Allocates use of a network device driver (NDD).

Syntax

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

int ns_alloc (nddname, nddpp)
       char * nddname;
       struct ndd ** nddpp;
```

Parameters

Item	Description
nddname	Specifies the device name to be allocated.
nddpp	Indicates the address of the pointer to a **ndd** structure.

Description

The **ns_alloc** network service searches the Network Service (NS) device chain to find the device driver with the specified *nddname* parameter. If the service finds a match, it increments the reference count for the specified device driver. If the reference count is incremented to 1, the **ndd_open** subroutine specified in the **ndd** structure is called to open the device driver.

Examples

The following example illustrates the **ns_alloc** network service:

```
struct ndd   *nddp;
error = ns_alloc("en0", &nddp);
```

Return Values

If a match is found and the **ndd_open** subroutine to the device is successful, a pointer to the **ndd** structure for the specified device is stored in the *nddpp* parameter. If no match is found or the open of the device is unsuccessful, a non-zero value is returned.

Item	Description
0	Indicates the operation was successful.
ENODEV	Indicates an invalid network device.
ENOENT	Indicates no network demuxer is available for this device.

The **ndd_open** routine may specify other return values.

Related reference:

"ns_free Network Service" on page 397

ns_attach Network Service

Purpose

Attaches a network device to the network subsystem.

Syntax

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

int ns_attach (nddp)
       struct ndd * nddp;
```

Parameters

Item	Description
nddp	Specifies a pointer to an **ndd** structure describing the device to be attached.

Description

The **ns_attach** network service places the device into the available network service (NS) device chain. The network device driver (NDD) should be prepared to be opened after the **ns_attach** network service is called.

Note: The **ndd** structure is allocated and initialized by the device. It should be pinned.

Examples

The following example illustrates the **ns_attach** network service:

```
struct ndd ndd;
ndd.ndd_name = "en0";
ndd.ndd_addrlen = 6;
ndd.ndd_hdrlen = 14;
ndd.ndd_mtu = ETHERMTU;
ndd.ndd_mintu = 60;
ndd.ndd_type = NDD_ETHER;
ndd.ndd_flags =
   NDD_BROADCAST | NDD_SIMPLEX;
ndd.ndd_open = entopen;
ndd.ndd_output = entwrite;
ndd.ndd_ctl = entctl;
ndd.ndd_close = entclose;
.
.
.
ns_attach(&ndd);
```

Return Values

Item	Description
0	Indicates the operation was successful.
EEXIST	Indicates the device is already in the available NS device chain.

Related reference:

"ns_detach Network Service" on page 397

ns_del_demux Network Service

Purpose

Deletes a demuxer for the specified type of network interface.

Syntax

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

int ns_del_demux (ndd_type)
       u_long  ndd_type;
```

Parameters

Item	Description
ndd_type	Specifies the network interface type of the demuxer that is to be deleted.

Description

If the demuxer is not currently in use, the **ns_del_demux** network service deletes the specified demuxer from the list of available network demuxers. A demuxer is in use if a network device driver (NDD) is open for the demuxer.

Examples

The following example illustrates the **ns_del_demux** network service:

```
ns_del_demux(NDD_ISO88023);
```

Return Values

Item	Description
0	Indicates the operation was successful.
ENOENT	Indicates the demuxer of the specified type does not exist.

Related reference:

"ns_add_demux Network Kernel Service" on page 387

ns_del_filter Network Service

Purpose

Deletes a receive filter.

Syntax

```
#include <sys/cdli.h>
#include <sys/ndd.h>

int ns_del_filter (nddp, filter, len)
       struct ndd * nddp;
       caddr_t  filter;
       int  len;
```

Parameters

Item	Description
nddp	Specifies the **ndd** structure that this delete request is for.
filter	Specifies the pointer to the receive filter.
len	Specifies the length in bytes of the receive filter.

Description

The **ns_del_filter** network service deletes the receive filter from the corresponding network demuxer. This disables packet reception for packets that match the filter. The delete request is passed on to the **nd_del_filter** function of the demuxer for the specified network device driver (NDD).

Examples

The following example illustrates the **ns_del_filter** network service:

```
struct ns_8022 dl;

dl.filtertype = NS_LLC_DSAP_SNAP;
dl.dsap = 0xaa;
dl.orgcode[0] = 0x0;
dl.orgcode[1] = 0x0;
dl.orgcode[2] = 0x0;
dl.ethertype = 0x0800;
ns_del_filter(nddp, &dl, sizeof(dl));
```

Return Values

Item	Description
0	Indicates the operation was successful.

The network demuxer may supply other return values.

Related reference:

"ns_add_filter Network Service" on page 388

"ns_alloc Network Service" on page 392

ns_del_status Network Service

Purpose

Deletes a previously added status filter.

Syntax

```
#include <sys/cdli.h>
#include <sys/ndd.h>

int ns_del_status (nddp, statfilter, len)
       struct ndd * nddp;
       caddr_t  statfilter;
       int  len;
```

Parameters

Item	Description
nddp	Specifies the pointer to the **ndd** structure to which this delete request applies.
statfilter	Specifies the pointer to the status filter.
len	Specifies the length, in bytes, of the value of the *statfilter* parameter.

Description

The **ns_del_status** network service deletes a previously added status filter from the corresponding network demuxer. The delete request is passed on to the **nd_del_status** function of the demuxer for the specified network device driver (NDD). This network service disables asynchronous status notification from the specified device.

Examples

The following example illustrates the **ns_del_status** network service:

```
error = ns_add_status(nddp, &filter,
sizeof(filter));
```

Return Values

Item	Description
0	Indicates the operation was successful.

The network demuxer may supply other return values.

Related reference:

"ns_add_status Network Service" on page 391

ns_detach Network Service

Purpose

Removes a network device from the network subsystem.

Syntax

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

int ns_detach (nddp)
       struct ndd * nddp;
```

Parameters

Item	Description
nddp	Specifies a pointer to an **ndd** structure describing the device to be detached.

Description

The **ns_detach** service removes the **ndd** structure from the chain of available NS devices.

Examples

The following example illustrates the **ns_detach** network service:

```
ns_detach(nddp);
```

Return Values

Item	Description
0	Indicates the operation was successful.
ENOENT	Indicates the specified *ndd* structure was not found.
EBUSY	Indicates the network device driver (NDD) is currently in use.

Related reference:

"ns_attach Network Service" on page 393

ns_free Network Service

Purpose

Relinquishes access to a network device.

Syntax

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

void ns_free (nddp)
       struct ndd * nddp;
```

Parameters

Item	Description
nddp	Specifies the **ndd** structure of the network device that is to be freed from use.

Description

The **ns_free** network service relinquishes access to a network device. The **ns_free** network service also decrements the reference count for the specified **ndd** structure. If the reference count becomes 0, the **ns_free** network service calls the **ndd_close** subroutine specified in the **ndd** structure.

Examples

The following example illustrates the **ns_free** network service:

```
struct ndd *nddp
ns_free(nddp);
```

Files

Item	Description
net/cdli.c	

Related reference:

"ns_alloc Network Service" on page 392

p

The following kernel services begin with the with the letter p.

__pag_getid System Call

Purpose

Invokes the **kcred_getpagid** kernel service and returns the PAG identifier for that PAG name.

Syntax

```
int __pag_getid (name)
char *name;
```

Description

Given a PAG type name, the **__pag_getid** invokes the **kcred_getpagid** kernel service and returns the PAG identifier for that PAG name.

Parameters

Item	Description
name	A **char *** value which references a NULL-terminated string of not more than PAG_NAME_LENGTH_MAX characters.

Return Values

If successful, a value greater than or equal to 0 is returned and represents the PAG type. This value may be used in subsequent calls to other PAG system calls that require a *type* parameter on input. If unsuccessful, -1 is returned and the **errno** global variable is set to a value reflecting the cause of the error.

Error Codes

Item	Description
ENOENT	The *name* parameter doesn't refer to an existing PAG type.
ENAMETOOLONG	The *name* parameter refers to a string that is longer than PAG_NAME_LENGTH_MAX.

Related reference:

"__pag_getname System Call"

"__pag_setname System Call" on page 400

"kcred_getpagid Kernel Service" on page 245

__pag_getname System Call

Purpose

Retrieves the name of a PAG type.

Syntax

```
int __pag_getname (type, buf, size)
int type;
char *buf;
int size;
```

Description

The **__pag_getname** system call retrieves the name of a PAG type given its integer value by invoking the **kcred_getpagname** kernel service with the given parameters.

Parameters

Item	Description
type	A numerical PAG identifier.
buf	A **char *** value that points to an array at least PAG_NAME_LENGTH_MAX+1 bytes in length.
size	An **int** value that gives the size of *buf* in bytes.

Return Values

If successful, 0 is returned and the *buf* parameter contains the PAG name associated with the *type* parameter. If unsuccessful, -1 is returned and the **errno** global variable is set to a value reflecting the cause of the error.

Error Codes

Item	Description
EINVAL	The value of the *type* parameter is less than 0 or greater than the maximum PAG identifier.
ENOENT	There is no PAG associated with the *type* parameter.
ENOSPC	The value of the *size* parameter is insuffient to hold the PAG name and its terminating NULL character.

Related reference:

"__pag_getvalue System Call"

"__pag_setname System Call" on page 400

"kcred_getpagname Kernel Service" on page 246

__pag_getvalue System Call

Purpose

Invokes the **kcred_getpag** kernel service and returns the PAG value.

Syntax

```
int __pag_getvalue (type)
int type;
```

Description

Given a PAG type, the **__pag_getvalue** system call invokes the **kcred_getpag** kernel service and returns the PAG value for the value of the *type* parameter.

Parameters

Item	Description
type	An **int** value indicating the desired PAG.

Return Values

If successful, the value of the PAG (or 0 when there is no value for that PAG type) is returned. If unsuccessful, -1 is returned and the **errno** global variable is set to a value reflecting the cause of the error.

Error Codes

Item	Description
EINVAL	The *type* parameter is less than 0 or greater than the maximum PAG type value.
ENOENT	The *type* parameter doesn't reference and existing PAG type.

Note: It is not an error for a defined PAG to not have a value in the current process' credentials.

Related reference:

"__pag_getid System Call" on page 398

"__pag_setvalue System Call" on page 401

"kcred_getpagname Kernel Service" on page 246

__pag_setname System Call

Purpose

Invokes the **kcred_setpagname** kernel service and returns the PAG type identifier.

Syntax

```
int __pag_setname (name, flags)
char *name;
int flags;
```

Description

The **__pag_setname** system call invokes the **kcred_setpagname** kernel service to register the name of a PAG and returns the PAG type identifier. The value of the *func* parameter to **kcred_setpagname** will be NULL. The other parameters to this system call are the same as with the underlying kernel service. This system call requires the SYS_CONFIG privilege.

Parameters

Item	Description
name	A **char** * value giving the symbolic name of the requested PAG.
flags	Either PAG_UNIQUEVALUE or PAG_MULTIVALUED 1 .

Return Values

A return value greater than or equal to 0 is the PAG type associated with the *name* parameter. This value may be used with other PAG-related system calls which require a numerical PAG identifier. If unsuccessful, -1 is returned and the **errno** global variable is set to indicate the cause of the error.

Error Codes

Item	Description
ENOSPC	The PAG name table is full.
EEXIST	The named PAG type already exists in the table, and the *flags* and *func* parameters do not match their previous values.
EPERM	The calling process does not have the SYS_CONFIG privilege.

Related reference:

"__pag_getname System Call" on page 399

"__pag_setvalue System Call"

"kcred_setpagname Kernel Service" on page 251

__pag_setvalue System Call

Purpose

Invokes the **kcred_setpag** kernel service and sets the value of PAG type to *pag*.

Syntax

```
int __pag_setvalue (type, pag)
int type;
int pag;
```

Description

Given a PAG type and value, the **__pag_setvalue** system call invokes the **kcred_setpag** kernel service and sets the value of PAG type to *pag*. This system call requires the SET_PROC_DAC privilege.

Parameters

Item	Description
type	An **int** value indicating the desired PAG.
pag	An **int** value containing the new PAG value.

Return Values

If successful, 0 is returned. If unsuccessful, -1 is returned and the **errno** global variable is set to a value reflecting the cause of the error.

Error Codes

Item	Description
ENOENT	The *type* parameter doesn't reference an existing PAG type.
EINVAL	The value of *pag* is -1.
EPERM	The calling process lacks the appropriate privilege.

Related reference:

"__pag_getvalue System Call" on page 399

"__pag_setname System Call" on page 400

"kcred_setpagname Kernel Service" on page 251

panic Kernel Service

Purpose

Crashes the system.

Syntax

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

panic ( s)
char *s;
```

Parameter

Item	Description
s	Points to a character string to be written to the error log.

Description

The **panic** kernel service is called when a catastrophic error occurs and the system can no longer continue to operate. The **panic** service performs these two actions:

- Writes the character string pointed to by the *s* parameter to the error log.
- Performs a system dump.

The system halts after the dump. You should wait for the dump to complete, reboot the system, and then save and analyze the dump.

Execution Environment

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

Return Values

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

Related information:

RAS Kernel Services

pci_cfgrw Kernel Service

Purpose

Reads and writes PCI bus slot configuration registers.

Syntax

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

```
int pci_cfgrw(bid, md, write_flag)
int bid;
struct mdio *md;
int write_flag;
```

Description

The **pci_cfgrw** kernel service provides serialized access to the configuration registers for a PCI bus. To ensure data integrity in a multi-processor environment, a lock is required before accessing the configuration registers. Depending on the value of the *write_flag* parameter, a read or write to the configuration register is performed at offset *md_addr* for the device identified by *md_sla*.

The **pci_cfgrw** kernel service provides for kernel extensions the same services as the **MIOPCFGET** and **MIOPCFPUT** ioctls provides for applications. The **pci_cfgrw** kernel service can be called from either the process or the interrupt environment.

Parameters

Item	Description
bid	Specifies the bus identifier.
md	Specifies the address of the *mdio* structure. The *mdio* structure contains the following fields:

md_addr	Starting offset of the configuration register to access (**0** to **0xFF** for PCI/PCI-X, and 0 to 0xFFF for PCI-E).
md_data	Pointer to the data buffer.
md_size	Number of items of size specified by the *md_incr* parameter. The maximum size is 256 bytes for PCI/PCI-X, and 4096 for PCI-E.
md_incr	Access types, **MV_BYTE**, **MV_WORD**, or **MV_SHORT**.
md_sla	Device Number and Function Number. (Device Number * 8) + Function.

write_flag	Set to **1** for write and **0** for read.

Return Values

Returns 0 for successful completion.

Item	Description
ENOMEM	Indicates no memory could be allocated.
EINVAL	Indicated that the bus, device/function, or size is not valid.
EPERM	Indicates that the platform does not allow the requested operation

Related information:
Machine Device Driver

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

Item	Description
cmd	Specifies the command to pass on to protocols.
sa	Indicates the address of a **sockaddr** structure that is passed to the protocols.

Description

The **pfctlinput** kernel service searches through the protocol switch table of each configured domain and invokes the protocol **ctlinput** function if defined. Both the *cmd* and *sa* parameters are passed as parameters to the protocol function.

Execution Environment

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

Return Values

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

Related information:

Network Kernel Services

Understanding Socket Header Files

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

Item	Description
family	Specifies the address family for which to search.
protocol	Indicates the protocol within the address family.
type	Specifies the type of socket (for example, **SOCK_RAW**).

Description

The **pffindproto** kernel service first searches the domain switch table for the address family specified by the *family* parameter. If found, the **pffindproto** service then searches the protocol switch table for that domain and checks for matches with the *type* and *protocol* parameters.

If a match is found, the **pffindproto** service returns the address of the protocol switch table entry. If the *type* parameter is set to **SOCK_RAW**, the **pffindproto** service returns the first entry it finds with protocol equal to 0 and type equal to **SOCK_RAW**.

Execution Environment

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

Return Values

The **pffindproto** service returns a null value if a protocol switch table entry was not found for the given search criteria. Upon success, the **pffindproto** service returns the address of a protocol switch table entry.

Related information:

Network Kernel Services

Understanding Socket Header Files

pgsignal Kernel Service

Purpose

Sends a signal to all of the processes in a process group.

Syntax

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

void pgsignal ( pid,  sig)
pid_t pid;
int sig;
```

Parameters

Item	Description
pid	Specifies the process ID of a process in the group of processes to receive the signal.
sig	Specifies the signal to send.

Description

The **pgsignal** kernel service sends a signal to each member in the process group to which the process identified by the *pid* parameter belongs. The *pid* parameter must be the process identifier of the member of the process group to be sent the signal. The *sig* parameter specifies which signal to send.

Device drivers can get the value for the *pid* parameter by using the **getpid** kernel service. This value is the process identifier for the currently executing process.

The **sigaction** subroutine contains a list of the valid signals.

Execution Environment

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

Return Values

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

Related reference:

"getpid Kernel Service" on page 189

"pidsig Kernel Service"

Related information:

sigaction subroutine

Process and Exception Management Kernel Services

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

Item	Description
pid	Specifies the process ID of the receiving process.
sig	Specifies the signal to send.

Description

The **pidsig** kernel service sends a signal to a process. The *pid* parameter must be the process identifier of the process to be sent the signal. The *sig* parameter specifies the signal to send. See the **sigaction** subroutine for a list of the valid signals.

Device drivers can get the value for the *pid* parameter by using the **getpid** kernel service. This value is the process identifier for the currently executing process.

The **pidsig** kernel service can be called from an interrupt handler execution environment if the process ID is known.

Execution Environment

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

Return Values

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

Related reference:

"getpid Kernel Service" on page 189

"pgsignal Kernel Service" on page 405

Related information:

sigaction subroutine

Process and Exception Management Kernel Services

pin Kernel Service

Purpose

Pins the address range in the system (kernel) space.

Syntax

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

int pin ( addr,  length)
caddr_t  addr;
int length;
```

Parameters

Item	Description
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 **xmempin** 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

Item	Description
0	Indicates successful completion.
EINVAL	Indicates that the value of the *length* parameter is negative or 0. Otherwise, the area of memory beginning at the address of the first byte to pin (the *addr* parameter) and extending for the number of bytes specified by the *length* parameter is not defined.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOMEM	Indicates that the **pin** service was unable to pin due to insufficient real memory or exceeding the systemwide pin count.
ENOSPC	Indicates insufficient file system or paging space.

Related reference:

"xmempin Kernel Service" on page 604

"xmemunpin Kernel Service" on page 605

Related information:

Understanding Execution Environments

Memory Kernel Services

pin_context_stack or unpin_context_stack Kernel Service

Purpose

Pins and unpins hidden kernel stack region.

Syntax

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

kerrno_t pin_context_stack (flags)
long flags;

kerrno_t unpin_context_stack (flags)
long flags;
```

Parameters

Item	Description
flags	Various flags to the kernel service. Must be set to 0.

Description

Kernel code that pins its system call stack should call this service before the first kernel stack pin and call the **unpin_context_stack()** service after the last unpin. These services do not pin or unpin the C execution stack, but instead pin or unpin a hidden stack resource used for the kernel-key support.

Execution Environment

These services must be called in the process environment.

Return Values

Item	Description
0	Indicates a successful completion.
ENOMEM_PIN_CONTEXT_STACK	Indicates that the memory is not sufficient to satisfy the request.
ENOSPC_PIN_CONTEXT_STACK	Indicates that the page space is not sufficient.
EINVAL_PIN_CONTEXT_STACK	Indicates that the execution environment is not valid.
EINVAL_UNPIN_CONTEXT_STACK	Indicates that the execution environment is not valid. (For example, the service is not in the process environment or the kernel keys are not enabled or the value of the *flag* parameter is not valid.)

Related reference:

"vm_setseg_kkey Kernel Service" on page 569

"vm_protect_kkey Kernel Service" on page 559

"raschk_eaddr_kkey Kernel Service" on page 435

"xmgethkeyset Kernel Service" on page 611

"xmsethkeyset Kernel Service" on page 612

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

Item	Description
delta	Specifies the amount by which to change the number of free-pinned character buffers.

Description

The **pincf** service is used to control the size of the list of free-pinned character buffers. A positive value for the *delta* parameter increases the size of this list, while a negative value decreases the size.

All device drivers that use character blocks need to use the **pincf** service. These drivers must indicate with a positive delta value the maximum number of character blocks they expect to be using concurrently. Device drivers typically call this service with a positive value when the **ddopen** routine is called. They should call the **pincf** service with a negative value of the same amount when they no longer need the pinned character blocks. This occurs typically when the **ddclose** routine is called.

Execution Environment

The **pincf** kernel service can be called in the process environment only.

Return Values

The **pincf** service returns a value representing the amount by which the service changed the number of free-pinned character buffers.

Related reference:

"waitcfree Kernel Service" on page 582

Related information:

I/O Kernel Services

pincode Kernel Service

Purpose

Pins the code and data associated with a loaded object module.

Syntax

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

int pincode ( func)
int (*func) ();
```

Parameter

Item	Description
func	Specifies an address used to determine the object module to be pinned. The address is typically that of a function exported by this object module.

Description

The **pincode** service uses the **pin** service to pin the specified object module. The loader entry for the object module is used to determine the size of both the code and data.

Execution Environment

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

Return Values

Item	Description
0	Indicates successful completion.
EINVAL	Indicates that the *func* parameter is not a valid pointer to the function.
ENOMEM	Indicates that the **pincode** service was unable to pin the module due to insufficient real memory.

When an error occurs, the **pincode** service returns without pinning any pages.

Related reference:

"pin Kernel Service" on page 407

Related information:

Understanding Execution Environments

Memory Kernel Services

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

Item	Description
ioparms	Points to parameters for the I/O routine.
iofunc	Specifies the I/O routine function pointer.
iorecov	Specifies the I/O recovery routine function pointer.

Description

The **pio_assist** kernel service assists in handling exceptions caused by programmed I/O. Use of the **pio_assist** service standardizes the programmed I/O exception handling for all routines performing programmed I/O. The **pio_assist** service is built upon other kernel services that routines access to provide their own exception handling if the **pio_assist** service should not be used.

Using the pio_assist Kernel Service

To use the **pio_assist** service, the device handler writer must provide a callable routine that performs the I/O operation. The device handler writer can also optionally provide a routine that can recover and log I/O errors. The mainline device handler code would then call the **pio_assist** service with the following parameters:

- A pointer to the parameters needed by the I/O routine
- The function pointer for the routine performing I/O
- A pointer for the I/O recovery routine (or a null pointer, if there is no I/O recovery routine)

If the pointer for the I/O recovery routine is a null character, the *iofunc* routine is recalled to recover from I/O exceptions. The I/O routine for error retry should only be re-used if the I/O routine can handle being recalled when an error occurs, and if the sequence of I/O instructions can be reissued to recover from typical bus errors.

The *ioparms* parameter points to the parameters needed by the I/O routine. It is passed to the I/O routine when the **pio_assist** service calls the I/O routine. It is also passed to the I/O recovery routine when the I/O recovery routine is invoked by the **pio_assist** service. If any of the parameters found in the structure pointed to by the *ioparms* parameter are modified by the *iofunc* routine and needed by the *iorecov* or recalled *iofunc* routine, they must be declared as *volatile*.

Requirements for Coding the Caller-Provided I/O Routine

The *iofunc* parameter is a function pointer to the routine performing the actual I/O. It is called by the **pio_assist** service with the following parameters:

```
int iofunc (ioparms)
caddr_t ioparms;            /* pointer to parameters */
```

The *ioparms* parameter points to the parameters used by the I/O routine that was provided on the call to the **pio_assist** kernel service.

If the **pio_assist** kernel service is used with a null pointer to the *iorecov* I/O recovery routine, the *iofunc* I/O routine is called to retry all programmed I/O exceptions. This is useful for devices that have I/O operations that can be re-sent without concern for hardware state synchronization problems.

Upon return from the I/O, the return code should be 0 if no error was encountered by the I/O routine itself. If a nonzero return code is presented, it is used as the return code from the **pio_assist** kernel service.

Requirements for Coding the Caller-Provided I/O Recovery Routine

The *iorecov* parameter is a function pointer to the device handler's I/O recovery routine. This *iorecov* routine is responsible for logging error information, if required, and performing the necessary recovery operations to complete the I/O, if possible. This may in fact include calling the original I/O routine. The *iorecov* routine is called with the following parameters when an exception is detected during execution of the I/O routine:

```
int iorecov (parms, action, infop)
caddr_t parms;/* pointer to parameters passed to iofunc*/
int action;           /* action indicator */
struct pio_except *infop;       /* pointer to exception info */
```

The *parms* parameter points to the parameters used by the I/O routine that were provided on the call to the **pio_assist** service.

The *action* parameter is an operation code set by the **pio_assist** kernel service to one of the following:

Item	Description
PIO_RETRY	Log error and retry I/O operations, if possible.
PIO_NO_RETRY	Log error but do not retry the I/O operation.

The **pio_except** structure containing the exception information is platform-specific and defined in the **/usr/include/sys/except.h** file. The fields in this structure define the type of error that occurred, the bus address on which the error occurred, and additional platform-specific information to assist in the handling of the exception.

The *iorecov* routine should return with a return code of 0 if the exception is a type that the routine can handle. A **EXCEPT_NOT_HANDLED** return code signals that the exception is a type not handled by the *iorecov* routine. This return code causes the **pio_assist** kernel service to invoke the next exception handler on the stack of exception handlers. Any other nonzero return code signals that the *iorecov* routine handled the exception but could not successfully recover the I/O. This error code is returned as the return code from the **pio_assist** kernel service.

Return Codes by the pio_assist Kernel Service

The **pio_assist** kernel service returns a return code of 0 if the *iofunc* I/O routine does not indicate any errors, or if programmed I/O exceptions did occur but were successfully handled by the *iorecov* I/O recovery routine. If an I/O exception occurs during execution of the *iofunc* or *iorecov* routines and the exception count has not exceeded the maximum value, the *iorecov* routine is called with an *op* value of **PIO_RETRY**.

If the number of exceptions that occurred during this operation exceeds the maximum number of retries set by the platform-specific value of **PIO_RETRY_COUNT**, the **pio_assist** kernel service calls the *iorecov* routine with an *op* value of **PIO_NO_RETRY**. This indicates that the I/O operation should not be retried. In this case, the **pio_assist** service returns a return code value of **EIO** indicating failure of the I/O operation.

If the exception is not an I/O-related exception or if the *iorecov* routine returns with the return code of **EXCEPT_NOT_HANDLED** (indicating that it could not handle the exception), the **pio_assist** kernel service does not return to the caller. Instead, it invokes the next exception handler on the stack of exception handlers for the current process or interrupt handler. If no other exception handlers are on the stack, the default exception handler is invoked. The normal action of the default exception handler is to cause a system crash.

Execution Environment

The **pio_assist** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
0	Indicates that either no errors were encountered, or PIO errors were encountered and successfully handled.
EIO	Indicates that the I/O operation was unsuccessful because the maximum number of I/O retry operations was exceeded.

Related information:

Kernel Extension and Device Driver Management Kernel Services

User-Mode Exception Handling

Process State-Change Notification Routine

Purpose

Allows kernel extensions to be notified of major process and thread state transitions.

Syntax

```
void prochadd_handler ( term,  type,  id)
struct proch *term;
int type;
long id;

void proch_reg_handler ( term,  type,  id)
struct prochr *term;
int type;
long id;
```

Parameters

Item	Description
term	Points to the **proch** structure used in the **prochadd** call or to the **prochr** structure used in the **proch_reg** call.

Item **Description**

type
Defines the state change event being reported: process initialization, process termination, process exec, thread initialization, or thread termination. These values are defined in the **/usr/include/sys/proc.h** file. The values that may be passed as *type* also depend on how the callout is requested.

Possible **prochadd_handler** *type* values:

PROCH_INITIALIZE
Process is initializing.

PROCH_TERMINATE
Process is terminating.

PROCH_EXEC
Process is about to exec a new program.

THREAD_INITIALIZE
A new thread is created.

THREAD_TERMINATE
A thread is terminated.

Possible **proch_reg_handler** *type* values:

PROCHR_INITIALIZE
Process is initializing.

PROCHR_TERMINATE
Process is terminating.

PROCHR_EXEC
Process is about to exec a new program.

PROCHR_THREAD_INIT
A new thread is created.

PROCHR_THREAD_TERM
A thread is terminated.

id
Defines either the process ID or the thread ID.

Description

The notification callout is set up by using either the **prochadd** or the **proch_reg** kernel service. If you request the notification using the **prochadd** kernel service, the callout follows the syntax shown first as **prochadd_handler**. If you request the notification using the **proch_reg** kernel service, the callout follows the syntax shown second as **proch_reg_handler**.

For process initialization, the **process state-change notification** routine is called in the execution environment of a parent process for the initialization of a newly created child process. For kernel processes, the notification routine is called when the **initp** kernel service is called to complete initialization.

For process termination, the notification routines are called before the kernel handles default termination procedures. 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.

Related reference:

"prochadd Kernel Service" on page 416

"prochdel Kernel Service" on page 418

Related information:

Kernel Extension and Device Driver Management Kernel Services

proch_reg Kernel Service

Purpose

Registers a callout handler.

Syntax

```
#include <sys/proc.h>
int proch_reg(struct prochr *)
```

Note: The prochr structure contains the following elements that must be set prior to calling **proch_reg**:

```
void (* proch_handler)(struct prochr *, int, long)
unsigned int  int prochr_mask
```

Parameters

Item	Description
int prochr_mask	Specifies the set of kernel events for which a callout is requested. Unlike the old_style interface, the callout is invoked only for the specified events. This mask is formed by ORing together any of these defined values: **PROCHR_INITIALIZE** Process created. **PROCHR_TERMINATE** Process terminated **PROCHR_EXEC** Process has issued the exec system call **PROCHR_THREADINIT** Thread created **PROCHR_THREADTERM** Thread terminated
proch_handler	Specifies the callout function to be called when specified kernel events occur.

Description

If the same **struct prochr *** is registered more than once, only the most recently specified information is retained in the kernel.

The **struct prochr *** is not copied to a new location in memory. As a result, if the structure is changed, results are unpredictable. This structure does not need to be pinned.

The primary consideration for the new-style interface is to improve scalability. A lock is only acquired when callouts are made. A summary mask of all currently registered callout event types is maintained. This summary mask is updated every time **proch_reg** or **proch_unreg** is called, even when registering an identical **struct prochr ***. Further, the lock is a complex lock, so once callouts have been registered, there is no lock contention in invoking them because the lock is held read-only.

When a callout to a registered handler function is made, the parameters passed are:

- a pointer to the registered prochr structure
- a callout request value to indicate the reason for the callout
- a thread or process ID

Return Values

On successful completion, the **proch_reg** kernel service returns a value of 0. The only error (non-zero) return is from trying to register with a NULL pointer.

Execution Environment

The **proch_reg** kernel service can be called from the process environment only.

Related reference:

"proch_unreg Kernel Service"

"Process State-Change Notification Routine" on page 413

Related information:

Kernel Extension and Driver Management Kernel Services

proch_unreg Kernel Service

Purpose

Unregisters a callout handler that was previously registered using the **proch_reg** kernel service.

Syntax

```
#include <sys/proc.h>
int proch_unreg(struct prochr *old_prochr);
```

Parameter

Item	Description
old_prochr	Specifies the address of the **proch** structure to be unregistered.

Description

Unregisters an existing callout handler that was previously registered using the **proch_reg()** kernel service.

Return Values

On successful completion, the **proch_unreg** kernel service returns a value of 0. An error (non-zero) return occurs when trying to unregister a handler that is not presently registered.

Execution Environment

The **proch_unreg** kernel service can be called from the process environment only.

Related reference:

"proch_reg Kernel Service" on page 415

Related information:

Kernel Extension and Driver Management Kernel Services

prochadd Kernel Service

Purpose

Adds a system-wide process state-change notification routine.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/proc.h>

void prochadd ( term)
struct proch *term;
```

Parameters

Item	Description
term	Points to a **proch** structure containing a notification routine to be added from the chain of systemwide notification routines.

Description

The **prochadd** kernel service allows kernel extensions to register for notification of major process state transitions. The **prochadd** service allows the caller to be notified when a process:

- Has just been created.
- Is about to be terminated.
- Is executing a new program.

The complete list of callouts is:

Callout	Description
PROCH_INITIALIZE	Process (pid) created (**initp**, **kforkx**)
PROCH_TERMINATE	Process (pid) terminated (**kexitx**)
PROCH_EXEC	Process (pid) executing (**execvex**)
THREAD_INITIALIZE	Thread (tid) created (**kforkx**, **thread_create**)
THREAD_TERMINATE	Thread (tid) created (**kexitx**, **thread_terminate**)

The **prochadd** service is typically used to allow recovery or reassignment of resources when processes undergo major state changes.

The caller should allocate a **proch** structure and update the `proch.handler` field with the entry point of a caller-supplied notification routine before calling the **prochadd** kernel service. This notification routine is called once for each process in the system undergoing a major state change.

The **proch** structure has the following form:

```
struct proch
{
        struct proch *next
        void                 *handler ();
}
```

Execution Environment

The **prochadd** kernel service can be called from the process environment only.

Related reference:

"prochdel Kernel Service" on page 418

"Process State-Change Notification Routine" on page 413

Related information:

Kernel Extension and Driver Management Kernel Services

prochdel Kernel Service

Purpose

Deletes a process state change notification routine.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/proc.h>

void prochdel ( term)
struct proch *term;
```

Parameter

Item	Description
term	Points to a **proch** structure containing a notification routine to be removed from the chain of system-wide notification routines. This structure was previously registered by using the **prochadd** kernel service.

Description

The **prochdel** kernel service removes a process change notification routine from the chain of system-wide notification routines. The registered notification routine defined by the handler field in the **proch** structure is no longer to be called by the kernel when major process state changes occur.

If the **proch** structure pointed to by the *term* parameter is not found in the chain of structures, the **prochdel** service performs no operation.

Execution Environment

The **prochdel** kernel service can be called from the process environment only.

Related reference:

"prochadd Kernel Service" on page 416

"Process State-Change Notification Routine" on page 413

Related information:

Kernel Extension and Driver Management Kernel Services

probe or kprobe Kernel Service

Purpose

Logs errors with symptom strings.

Library (for probe)

Run-time Services Library.

Syntax

```
#include <sys/probe.h>
or
#include <sys/sysprobe.h>
int probe ( probe_p)
probe_t *probe_p
int kprobe (probe_p)
probe_t *probe_p
```

Description

The probe subroutine logs an entry to the error log. The entry consists of an error log entry as defined in the **errlog** subroutine and the **err_rec.h** header file, and a symptom string.

The **probe** subroutine is called from an application, while **kprobe** is called from the Kernel and Kernel extensions. Both **probe** and **kprobe** have the same interfaces, except for return codes.

IBM software should use the **sys/sysprobe.h** header file while non-IBM programs should include the **sys/probe.h file**. This is because IBM symptom strings must conform to different rules than non-IBM strings. It also tells any electronic support application whether or not to route the symptom string to IBM's Retain database.

Parameters

Item	Description
probe_p	is a pointer to the data structure which contains the pointer and length of the error record, and the data for the probe. The error record is described under the **errlog** subroutine and defined in **err_rec.h**. The first word of the structure is a magic number to identify this version of the structure. The magic number should be set to `PROBE_MAGIC`. **Note:** `PROBE_MAGIC` is different between **probe.h** and **sysprobe.h** to distinguish an IBM symptom string from a non-IBM string. The probe data consists of flags which control probe handling, the number of symptom string keywords, followed by an array consisting of one element for each keyword.

Flags

Item	Description
SSNOSEND	indicates this symptom string shouldn't be forwarded to automatic problem opening facilities. An example where **SSNOSEND** should be used is in symptom data used for debugging purposes.
nsskwd	This gives the number of keywords specified (i.e.), the number of elements in the sskwds array.

Item **Description**

sskwds This is an array of keyword/value pairs. The keywords and their values are in the following table. The **I/S** value indicates whether the *keyword* and *value* are informational or are part of the logged symptom string. The number in parenthesis indicates, where applicable, the maximum string length.

```
keyword          I/S   value   type   Description

SSKWD_LONGNAME   I     char *  (30)   Product's long name
SSKWD_OWNER      I     char *  (16)   Product's owner
SSKWD_PIDS       S     char *  (11)   product id.
(required for IBM symptom strings)
SSKWD_LVLS       S     char *  (5)    product level
                   (required for IBM symptom strings)
SSKWD_APPLID     I     char *  (8)    application id.
SSKWD_PCSS       S     char *  (8)    probe id
                   (required for all symptom strings)
SSKWD_DESC       I     char *  (80)   problem description
SSKWD_SEV        I     int            severity from
                   1 (highest) to 4 (lowest).
                    3 is the default.
SSKWD_AB         S     char *  (5)    abend code
SSKWD_ADRS       S     void *         address. If used at all,
                   this should be a relative address.
SSKWD_DEVS       S     char *  (6)    Device type
SSKWD_FLDS       S     char *  (9)    arbitrary character string.
                   This is usually a field name and
                                      the SSKWD_VALUE
                   keyword specifies the value.
SSKWD_MS         S     char *  (11)   Message number
SSKWD_OPCS       S     char *  (8)    OP code
SSKWD_OVS        S     char *  (9)    overwritten storage
SSKWD_PRCS       S                    unsigned long return code
SSKWD_REGS       S     char *  (4)    Register name (e.g.)
                   GR15 or LR unsigned long Value
SSKWD_VALU       S
SSKWD_RIDS       S     char *  (8)    resource or module id.
SSKWD_SIG        S .   int            Signal number
SSKWD_SN         S     char *  (7)    Serial Number
SSKWD_SRN        S     char *  (9)    Service Req. Number If specified,
                   and no error is logged,
                                      a hardware error is assumed.
SSKWD_WS         S     char *  (10)   Coded wait
```

Note: The **SSKWD_PCCS** value is always required. This is the probe id. Additionally, for IBM symptom strings, the **SSKWD_PIDS** and **SSKWD_LVLS** keywords are also required

If either the **erecp** or **erecl** fields in the **probe_rec** structure is 0 then no error logging record is being passed, and one of the default templates for symptom strings is used. The default template indicating a software error is used unless the **SSKWD_SRN** keyword is specified. If it is, the error is assumed to be a hardware error. If you don't want to log your own error with a symptom string, and you want to have a hardware error, and don't want to use the **SSKWD_SRN** value, then you can supply an error log record using the error identifier of **ERRID_HARDWARE_SYMPTOM**, see the **/usr/include/sys/errids.h** file.

Return Values for probe Subroutine

Item	Description
0	Successful
-1	Error. The errno variable is set to
EINVAL	Indicates an invalid parameter
EFAULT	Indicates an invalid address

Return Values for kprobe Kernal Service

Item	Description
0	Successful
EINVAL	Indicates an invalid parameter

Execution Environment

probe is executed from the application environment.

kprobe is executed from the Kernel and Kernel extensions. Currently, **kprobe** must not be called with interrupts disabled.

Files

Item	Description
/usr/include/sys/probe.h	Contains parameter definition.

Related reference:

"errsave or errlast Kernel Service" on page 138

Related information:

Error Logging Overview

errlog subroutine

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

Item	Description
dev	Specifies the device containing the block to be purged.
blkno	Specifies the block to be purged.

Description

The **purblk** kernel service purges (that is, makes unreclaimable by marking the block with a value of **STALE**) the specified block from the buffer cache.

Execution Environment

The **purblk** kernel service can be called from the process environment only.

Return Values

The **purblk** service has no return values.

Related reference:

"brelse Kernel Service" on page 32

"geteblk Kernel Service" on page 185

Related information:

Block I/O Buffer Cache Kernel Services: Overview

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

Item	Description
c	Specifies the character to place on the character list.
header	Specifies the address of the **clist** structure that describes the character list.

Description

> **Attention:** The caller of the **putc** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character blocks acquired from the **getcf** service are also pinned. Otherwise, the system may crash.

The **putc** kernel service puts the character specified by the *c* parameter at the end of the character list pointed to by the *header* parameter.

If the **putc** service indicates that there are no more buffers available, the **waitcfree** service can be used to wait until a character block is available.

Execution Environment

The **putc** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
0	Indicates successful completion.
-1	Indicates that the character list is full and no more buffers are available.

Related reference:

"getcb Kernel Service" on page 182

"putcf Kernel Service" on page 425

Related information:

I/O Kernel Services

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

Item	Description
p	Specifies the address of the character buffer to place on the character list.
header	Specifies the address of the **clist** structure that describes the character list.

Description

> **Attention:** The caller of the **putcb** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character blocks acquired from the **getcf** service are pinned. Otherwise, the system may crash.

The **putcb** kernel service places the character buffer pointed to by the *p* parameter on the end of the character list specified by the *header* parameter. Before calling the **putcb** service, you must load this new buffer with characters and set the `c_first` and `c_last` fields in the **cblock** structure. The *p* parameter is the address returned by either the **getcf** or the **getcb** service.

Execution Environment

The **putcb** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
0	Indicates successful completion.
-1	Indicates that the character list is full and no more buffers are available.

Related reference:

"getcb Kernel Service" on page 182

"putcf Kernel Service" on page 425

Related information:

I/O Kernel Services

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

Item	Description
header	Specifies the address of the **clist** structure that describes the character list.
source	Specifies the address from which characters are read to be placed on the character list.
n	Specifies the number of characters to be placed on the character list.

Description

> **Attention:** The caller of the **putcbp** service must ensure that the character list is pinned. This includes the **clist** header and all of the **cblock** character buffers. Character blocks acquired from the **getcf** service are pinned. Otherwise, the system may crash.

The **putcbp** kernel service operates on the characters specified by the *n* parameter starting at the address pointed to by the *source* parameter. This service places these characters at the end of the character list pointed to by the *header* parameter. The **putcbp** service then returns the number of characters added to the character list. If the character list is full and no more buffers are available, the **putcbp** service returns a 0. Otherwise, it returns the number of characters written.

Execution Environment

The **putcbp** kernel service can be called from either the process or interrupt environment.

Return Values

The **putcbp** service returns the number of characters written or a value of 0 if the character list is full, and no more buffers are available.

Related reference:

"pincf Kernel Service" on page 409

"putcf Kernel Service"
"waitcfree Kernel Service" on page 582

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

Item	Description
p	Identifies which character buffer to free.

Description

The **putcf** kernel service unpins the indicated character buffer.

The **putcf** service returns the specified buffer to the list of free character buffers.

Execution Environment

The **putcf** kernel service can be called from either the process or interrupt environment.

Return Values

The **putcf** service has no return values.

Related information:
I/O Kernel Services

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

Item	Description
header	Identifies which list of character buffers to free.

Description

The **putcfl** kernel service returns the specified list of buffers to the list of free character buffers. The **putcfl** service unpins the indicated character buffer.

Note: The caller of the **putcfl** service must ensure that the header and **clist** structure are pinned.

Execution Environment

The **putcfl** kernel service can be called from either the process or interrupt environment.

Return Values

The **putcfl** service has no return values.

Related information:

I/O Kernel Services

putcx Kernel Service

Purpose

Places a character on a character list.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/cblock.h>

int putcx ( c,  header)
char c;
struct clist *header;
```

Parameters

Item	Description
c	Specifies the character to place at the front of the character list.
header	Specifies the address of the **clist** structure that describes the character list.

Description

The **putcx** kernel service puts the character specified by the *c* parameter at the front of the character list pointed to by the *header* parameter. The **putcx** service is identical to the **putc** service, except that it puts the character at the front of the list instead of at the end.

If the **putcx** service indicates that there are no more buffers available, the **waitcfree** service can be used to wait until a character buffer is available.

Note: The caller of the **putcx** service must ensure that the character list is pinned. This includes the **clist** header and all the **cblock** character buffers. Character blocks acquired from the **getcf** service are pinned.

Execution Environment

The **putcx** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
0	Indicates successful completion.
-1	Indicates that the character list is full and no more buffers are available.

Related reference:

"pincf Kernel Service" on page 409

"putcfl Kernel Service" on page 425

Related information:

I/O Kernel Services

q

The following kernel services begin with the with the letter q.

query_proc_info Kernel Service

Purpose

Returns specific information about the current process or thread.

Syntax

```
#include <sys/encap.h>

int query_proc_info (type)
int type;
```

Parameters

Item	Description
type	Specifies the type of process or thread information requested. The *type* parameter can be one of the following values: **QPI_XPG_SUS_ENV** Queries whether the calling process has SPEC 1170 environment active. **QTI_FUNNELLED** Queries whether the current thread is funneled.

Description

The **query_proc_info** kernel service returns information about the current process or thread.

When called with the value QPI_XPG_SUS_ENV as the *type* parameter, it returns TRUE (1) when the process has SPEC 1170 active, that is, the process was issued with the environment variable XPG_SUS_ENV defined. Otherwise, the routine returns FALSE (0). When called with the value QTI_FUNNELLED as the *type* parameter, the **query_proc_info** kernel service returns TRUE (1) if the current thread has been funneled. Otherwise, the routine returns FALSE (0).

Execution Environment

The **query_proc_info** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
1	True.
0	False.

r

The following kernel services begin with the with the letter r.

RAS_BLOCK_NULL Exported Data Structure

Purpose

Allows for the silent failure of **ras_register** calls due to memory allocation errors.

Syntax

```
#include <sys/ras.h>

extern const ras_block_t RAS_BLOCK_NULL
```

Description

The **RAS_BLOCK_NULL** data structure allows components to go through their normal code paths when they receive an ENOMEM error from the **ras_register** kernel service. The presence of this data structure does not need to be explicitly checked by callers of RAS functions. All RAS domain functions (such as Component Tracing) are disabled with this control block.

Related reference:

"ras_register and ras_unregister Exported Kernel Services" on page 431

"ras_customize Exported Kernel Service" on page 429

Related information:

CT_HOOKx subroutine

ras_control Exported Kernel Service

Purpose

Controls component RAS characteristics.

Syntax

```
#include <sys/ras.h>

kerrno_t ras_control (
ras_block_t ras_blk,
ras_cmd_t command,
void * arg,
long argsize);
```

Description

The **ras_control** kernel service passes a command to the callback for the component referenced by the *ras_blk* parameter. If the *ras_blk* parameter is not known, use the **ras_path_control** call.

Note: During the **ras_control** process, callbacks to the registrant of the component might be initiated for changes that the RAS infrastructure makes to the component. The registrant should be aware of this for locking purposes (for instance, the registrant should not hold any locks that the callback needs).

If the *ras_blk* input parameter has a value of RAS_BLOCK_NULL, the **ras_control** kernel service returns without errors and takes no action.

Parameters

Item	**Description**
ras_blk	The target control block pointer.
command	Command passed to the callback. Commands are specific to a given RAS domain, such as Component Trace.
arg	Optional argument for the command.
argsize	Size of the argument, if a buffer or structure.

Execution Environment

The calling environment of the **ras_control** kernel service varies by individual command. The calling environment of a particular command is documented with the command itself.

Return Values

The **ras_control** kernel service returns 0 for success and a non-zero error code for failure.

Related reference:

"ras_customize Exported Kernel Service"

"ras_path_control Exported Kernel Services" on page 430

Related information:

Component Trace Facility

ras_callback subroutine

ras_customize Exported Kernel Service

Purpose

Loads persistent customized properties for a RAS control block.

Syntax

```
#include <sys/ras.h>

kerrno_t ras_customize (ras_block_t ras_blk);
```

Description

The **ras_customize** kernel service checks for, and applies persistent customized properties for a given *ras_blk* parameter. After applying any persistent properties, the **ras_customize** kernel service puts the *ras_blk* parameter in a usable state. Registration is not complete without a call to the **ras_customize** kernel service.

Note: During the **ras_customize** process, callbacks to the registrant might be initiated for changes that the RAS infrastructure makes to the component. The registrant should be aware of this for locking and initialization purposes (for example, the registrant should not be holding any locks that the callback needs, and the private data for the callback should be initialized before **ras_customize** is called).

If the *ras_blk* input parameter has a value of RAS_BLOCK_NULL, the **ras_customize** kernel service returns without errors and takes no action.

Parameters

Item	Description
ras_blk	The control block to act on. Must be previously allocated by the **ras_register** kernel service.

Execution Environment

The **ras_customize** kernel service must be called from the process environment.

Return Values

Item	Description
0	Successful.
non-zero	Unsuccessful.

Related reference:

"ras_control Exported Kernel Service" on page 428

Related information:

Component Trace Facility

ras_callback subroutine

ras_path_control Exported Kernel Services

Purpose

Controls component RAS characteristics.

Syntax

```
#include <sys/ras.h>

kerrno_t ras_path_control (
char * path,
ras_cmd_t command,
void * arg,
long argsize);
```

Description

The **ras_path_control** kernel service passes a command to the RAS component specified by the *path* parameter.

Note: During the **ras_path_control** process, callbacks to the registrant of the component might be initiated for changes that the RAS infrastructure makes to the component. The registrant should be aware of this for locking purposes (for instance, the registrant should not be holding any locks the callback needs).

Parameters

Item	Description
path	The pathname of the component to receive the *command* parameter.
command	Command passed to the callback. Commands are specific to a given RAS domain, such as Component Trace.
arg	Optional argument for the command.
argsize	Size of the argument, if a buffer or structure.

Execution Environment

The calling environment of the **ras_path_control** kernel service varies by individual command. The calling environment of a particular command is documented with the command itself.

Return Values

Item	Description
0	Successful.
non-zero	Unsuccessful.

Related reference:

"ras_control Exported Kernel Service" on page 428

"ras_customize Exported Kernel Service" on page 429

Related information:

Component Trace Facility

ras_register and ras_unregister Exported Kernel Services

Purpose

Registers and unregisters a RAS component.

Syntax

```
#include <sys/ras.h>

kerrno_t ras_register (
ras_block_t * rasbp,
char * name,
ras_block_t parent,
ras_type_t typesubtype,
char * desc,
long flags,
ras_callback_t ras_callback,
void * private_data);

kerrno_t ras_unregister (ras_block_t ras_blk);
```

Description

The **ras_register** kernel service and the **ras_unregister** kernel service register and unregister RAS handlers which are invoked by the kernel when the system needs to communicate various RAS commands to each component.

The **ras_register** kernel service registers a component with the given name under the *parent* provided. If the parent is NULL, the **ras_register** kernel service registers name as a base component, but the *typesubtype* parameter must be provided. The *name* parameter specifies the name for the subcomponent or base component (it is not a full component path). The *flags* field is used to specify what aspects of RAS the component understands. The *ras_callback* is the mechanism by which the RAS subsystem communicates various commands to the component, depending on what aspects of RAS the component understands. The *desc* parameter provides a short description for the component as a service aid.

The **ras_register** kernel service allocates a ras_block_t member and returns the control block for the component through the *rasbp* argument. This control block can be used in **ras_control** calls and further **ras_register** calls (to allocate children, for instance).

If the registration fails due to the system being out of memory, the value of the *rasbp* argument is set to RAS_BLOCK_NULL. All RAS functions for this component are disabled. RAS kernel services accept RAS_BLOCK_NULL control blocks but take no action. If the control block is set to RAS_BLOCK_NULLRAS, domain related functions (such as the **CT_HOOKx** and **CT_GEN** macros) run correctly but take no action. This action allows the ENOMEM type failures from the **ras_register** kernel service to be safely ignored. The value of the *rasbp* argument for all other types of errors is undefined.

The **ras_unregister** kernel service unregisters a component previously registered with the **ras_register** kernel service. The *ras_blk* parameter should have no further children.

Parameters

Item	Description
rasbp	The newly allocated ras_block_t member.
name	The name of the component, not its full pathname. Individual node names are limited to the number of characters specified by the value of the **RAS_NAME_MAX** parameter (including the terminating NULL character). The full component path (the concatenated names of a child component and all of its ancestors) is limited to the number of characters specified by the value of the **RAS_PATH_MAX** parameter (including the terminating NULL character). The **ras_register** kernel service reconstructs the full component path and rejects registrations for components whose full path exceeds the value of the **RAS_PATH_MAX** parameter. Node names are restricted to the character set "A-Z","a-z","0-9" and "_".
parent	An optional pointer to the parent component or NULL if none.
typesubtype	If parent is NULL, mandatory parameter is used to categorize the component. The top 16-bits of the lower word of this field are the type, and the bottom 16-bits are the subtype. The *typesubtype* is a ras_type_t member, which is an enum. See the **sys/ras_base.h** file for a description of the types available. If parent is non-NULL, this parameter is required to be the value of the **RAS_TYPE_CHILD** parameter.
desc	A short description string for the component. The *desc* string is limited to the number of characters specified by the value of the **RAS_DESC_MAX** parameter (including the terminating null). The *desc* string has no character set restriction. Any static elements of the string should be in U.S. English, but dynamic elements have no restriction.
flags	Indicates what type of RAS systems this component is aware of. Valid choices are the following: • **RASF_TRACE_AWARE**: Component is Component Trace aware. • **RASF_ERROR_AWARE**: Component is Error Checking aware. These flags are defined in the **sys/ras.h** file.
ras_callback	A function pointer provided by the registrant and called by the framework each time an external event modifies a property of the component. See the **ras_callback** interface specification.
private_data	An optional pointer to a component private memory area passed to the **ras_callback** function upon callback.
ras_blk	The control block to remove.

Execution Environment

Both the **ras_register** kernel service and the **ras_unregister** kernel service must be called from the process environment.

Return Values

The following are the return values of the **ras_register** kernel service.

Item	Description
0	Successful.
non-zero	Unsuccessful.

The following are the return values of the **ras_unregister** kernel service.

Item	Description
0	Successful.
non-zero	Unsuccessful.

Related reference:

"ras_customize Exported Kernel Service" on page 429

Related information:

Component Trace Facility

ras_callback subroutine

ras_ret_query_parms Kernel Service

Purpose

Returns callback parameters in the **ras_query_parms** structure.

Syntax

```
#include <sys/ras.h>

kerrno_t ras_ret_query_parms (retp, fmtstr, numstrings, descr)
ras_query_parms_t *retp;
char *fmtstr;
int numstrings;
char *descr[];
```

Parameters

Item	Description
retp	Points to the **ras_query_parms_t** data item to be filled in.
fmtstr	This is a format specifier. It has the following form: *spec-list* or `kywd=spec-list kywd=spec-list ...` Where the *spec-list* variable is of the form: *spec,spec,...* . The*spec* variable must be %x, %xx, %d, %dd, %s, or %ss. If the characters x, d, or s are doubled, for example, %xx, this indicates that multiple values are allowed. The following are some valid *fmtstr* values: **%x** One hexidecimal value. **%x,%d** One hexadecimal and one decimal value. **%xx** Multiple hexadecimal values. **k1=%x,%d k2=%dd** Keyword k1 takes one hexadecimal value and one decimal value. Keyword k2 takes multiple decimal values.
numstrings	Specifies the number of strings in the *descr* string array. The value must be at least 1.

Item	Description
descr	Specifies the component and parameters. There must be at least one string. The first string describes the component's function. If the component takes positional parameters, the following string(s) describe those. If keyword parameters are supplied, each keyword must have a corresponding *descr* string in the array describing that keyword. The **ras_ret_query_parms** kernel service does not return an error if the number of the *descr* strings does not match the format string. Instead, either the last keywords do not have help text, or the excess help strings are simply displayed.

Description

The **ras_ret_query_parms** kernel service can be used by a callback to aid in filling in the **ras_query_parms_t** structure when it receives the RASC_QUERY_PARMS call. This function formats the help text and places it into the **ras_query_parms_t** structure. If there is insufficient space for the help text in the provided **ras_query_parms_t** item, it returns **ENOMEM_RASC_CONTROL_QUERYPARMS**. The callback then just returns this error code.

The help text provided must follow the following conventions:

```
component - first line of description
component:parameters - parameter(s) description
```

or

```
component - first line of description
component:kywd1=parms - kywd1:parms description
component:kywd2=parms - kywd2:parms description
```

Execution Environment

The **ras_ret_query_parms** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful completion.
EINVAL_RAS_CONTROL_QUERYPARMS	Indicates that one or more parameters was not valid.
EFAULT_RAS_CONTROL_QUERYPARMS	Indicates that one or more parameter addresses was not valid.
ENOMEM_RAS_CONTROL_QUERYPARMS	Indicates that the rqp_text size was not large enough.

Related reference:

"dmp_compspec and dmp_compext Kernel Services" on page 92

raschk_eaddr_hkeyset Kernel Service

Purpose

Checks if an effective address can be referenced with a hardware keyset.

Syntax

```
#include <sys/raschk.h>
#include <sys/skeys.h>
#include <sys/kerrno.h>

kerrno_t rashchk_eaddr_hkeyset (eaddr, hset, flags)
void * eaddr;
hkeyset_t hset;
unsigned long flags;
```

Parameters

Item	Description
eaddr	Effective address to validate. Only one byte is checked.
hset	Hardware keyset to validate against.
flags	The following flags are defined:

RCHK_EHK_NOFAULT
No page faults are permitted while performing this check.

RCHK_EHK_NOPAGEIN
No page in is performed during this check.

RCHK_EHK_READ
Validates for read access.

RCHK_EHK_WRITE
Validates for write access.

Description

The **raschk_eaddr_hkeyset** kernel service performs an advisory runtime check to determine if an effective address can be referenced with a hardware keyset.

Read and write access checks are independently specified in the *flags* field. A check for read and write access requires both flags to be set.

Execution Environment

The **raschk_eaddr_hkeyset** kernel service can be called from the process or interrupt environment.

Return Values

Item	Description
0	Successful.
EFAULT_RASCHK_EADDR_HKEYSET	Operation failed because a page in or page fault was not allowed.
EFAULT_RASCHK_EADDR_HKEYSET_PROT	The address failed the protection check.
EINVAL_RASCHK_EADDR_HKEYSET	The address to validate was determined to be invalid, or neither READ nor WRITE checking was requested.

Related reference:
"raschk_eaddr_kkey Kernel Service"

raschk_eaddr_kkey Kernel Service

Purpose

Checks if an effective address can be referenced with a kernel-key.

Syntax

```
#include <sys/raschk.h>
#include <sys/kerrno.h>

kerrno_t raschk_eaddr_kkey (eaddr, kkey, flags)
void * eaddr;
kkey_t kkey;
unsigned long flags;
```

Parameters

Item	Description
eaddr	Effective address to validate. Only one byte is checked.
kkey	Kernel-key to check.
flags	The following flags are defined: **RCHK_EK_NOFAULT** No page faults of any kind are permitted while performing this check. **RCHK_EK_NOPAGEIN** No page in will be performed during this check.

Description

The **raschk_eaddr_kkey** kernel service performs an advisory runtime check to determine if an effective address can be referenced with a kernel-key. Note that read/write attributes are not maintained at a page granularity. This service only checks if the kernel-key assigned to an effective address matches the *kkey* value.

Execution Environment

The **raschk_eaddr_kkey** kernel service can be called from the process or interrupt environment.

Return Values

Item	Description
0	Successful.
EFAULT_RASCHK_EADDR_KKEY	Operation cannot be performed because a page in or page fault was not allowed.
EINVAL_RASCHK_EADDR_KKEY	The address to validate was determined to be invalid.
EINVAL_RASCHK_EADDR_KKEY_PROT	The address failed the protection check.

Related reference:

"raschk_eaddr_hkeyset Kernel Service" on page 434

raschk_stktrace Kernel Service

Purpose

Generates a runtime compact stack trace for only call chain addresses.

Syntax

```
#include <sys/raschk.h>

kerrno_t rashchk_stktrace (trcbufsz, flags, trcbuf)
size_t trcbufsz;
long flags;
void * trcbuf;
```

Parameters

Item	Description
trcbufsz	Size of the stack trace buffer the caller allocated.
flags	The following flags are defined: **RAS_STK_DO_CURMST** If this flag bit value is set, this service will not look at the previous MST to get the stack trace. The stack trace is obtained only for the current context. **RAS_STK_DO_PREVMST** If this flag bit value is set, this service will skip the current MST and start getting the stack trace from the previous MST. **RAS_STK_DO_ONEMST** This flag bit value can be combined with the above bit values to get stack trace for that MST. **RAS_STK_GET_SYMBOLS** If this flag bit value is set, then all the call chain addresses are translated into a stream of bytes containing symbols with offset (null terminated) and placed in the caller's buffer. **RAS_STK_DO_CURRWA** If this flag bit value is set, this service will use the RWA (recovery work area) associated with the current MST to begin the trace back. **Note:** The **RAS_STK_DO_PREVMST**, **RAS_STK_DO_CURMST**, and **RAS_STK_DO_CURRWA** flags are mutually exclusive. Specifying the **RAS_STK_DO_ONEMST** flag without specifying the **RAS_STK_DO_PREVMST** flag is equivalent to specifying the **RAS_STK_DO_CURMST** flag. If the **RAS_STK_GET_SYMBOLS** flag is not set, the end of the stack trace is indicated by an entry containing 0. A value of -2 in *trcbuf* indicates the start of a new **mst** trace if any. Also, the stack trace will stop once we reach the system call boundary as we are interested only in kernel stack trace and we can only validate kernel stack addresses. If the **RAS_STK_GET_SYMBOLS** flag is set, the output buffer will contain a null-terminated string with the symbolic representation of the stack trace. A call to **raschk_addr2sym()** is performed for each entry in the stack trace and the resulting strings are concatenated in the output buffer, and separated by '\n' characters. Special values in the stack trace will be translated to appropriate strings.
trcbuf	Pointer to the buffer that the caller allocated to get stack trace. **Note:** Ensure that *trcbuf* is pinned when called disabled.

Description

This kernel service can be used to generate a runtime compact stack trace. The algorithm is performed for:

- All MSTs starting from the current MST (default, and none of **RAS_STK_DO_CURMST**, **RAS_STK_DO_PREVMST**, **RAS_STK_DO_CURRWA**, nor **RAS_STK_DO_ONEMST** flag bits specified.)
- Only for the current MST (**RAS_STK_DO_CURMST** bit flag is set)
- All the MSTs starting from previous MST (**RAS_STK_DO_PREVMST** bit flag is set)
- Only for the previous MST (**RAS_STK_DO_PREVMST** and **RAS_STK_DO_ONEMST** bits are set)
- For the current MST recovery work area (RWA) context and previous MSTs. (**RAS_STK_DO_CURRWA** flag bit is set.)
- Only for the current MST recovery work area (RWA) context. (**RAS_STK_DO_CURRWA** and **RAS_STK_DO_ONEMST** flag bits are set.)
- Getting all the symbols plus offset corresponding to the call addresses obtained in *trcbuf* and replacing *trcbuf* with symbol information in a string format. (**RAS_STK_GET_SYMBOLS** bit flag is set)

Execution Environment

The **raschk_stktrace** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
0	Successful
kerrno	Unsuccessful

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

Item	Description
m0	Specifies the address of an **mbuf** structure containing input data.
proto	Specifies the protocol definition of data.
src	Identifies the **sockaddr** structure indicating where data is from.
dst	Identifies the **sockaddr** structure indicating the destination of the data.

Description

The **raw_input** kernel service accepts an input packet, builds a **raw_header** structure (as defined in the **/usr/include/net/raw_cb.h** file), and passes both on to the raw protocol input handler.

Execution Environment

The **raw_input** kernel service can be called from either the process or interrupt environment.

Return Values

The **raw_input** service has no return values.

Related information:

Network Kernel Services

raw_usrreq Kernel Service

Purpose

Implements user requests for raw protocols.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void raw_usrreq (so, req, m, nam, control)
struct  socket * so;
int  req;
struct  mbuf * m;
struct  mbuf * nam;
struct  mbuf * control;
```

Parameters

Item	Description
so	Identifies the address of a raw socket.
req	Specifies the request command.
m	Specifies the address of an **mbuf** structure containing data.
nam	Specifies the address of an **mbuf** structure containing the **sockaddr** structure.
control	This parameter should be set to a null value.

Description

The **raw_usrreq** kernel service implements user requests for the raw protocol.

The **raw_usrreq** service supports the following commands:

Command	Description
PRU_ABORT	Aborts (fast DISCONNECT, DETACH).
PRU_ACCEPT	Accepts connection from peer.
PRU_ATTACH	Attaches protocol to up.
PRU_BIND	Binds socket to address.
PRU_CONNECT	Establishes connection to peer.
PRU_CONNECT2	Connects two sockets.
PRU_CONTROL	Controls operations on protocol.
PRU_DETACH	Detaches protocol from up.
PRU_DISCONNECT	Disconnects from peer.
PRU_LISTEN	Listens for connection.
PRU_PEERADDR	Fetches peer's address.
PRU_RCVD	Have taken data; more room now.
PRU_RCVOOB	Retrieves out of band data.
PRU_SEND	Sends this data.
PRU_SENDOOB	Sends out of band data.
PRU_SENSE	Returns status into m.
PRU_SOCKADDR	Fetches socket's address.
PRU_SHUTDOWN	Will not send any more data.

Any unrecognized command causes the **panic** kernel service to be called.

Execution Environment

The **raw_userreq** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
EOPNOTSUPP	Indicates an unsupported command.
EINVAL	Indicates a parameter error.
EACCES	Indicates insufficient authority to support the **PRU_ATTACH** command.
ENOTCONN	Indicates an attempt to detach when not attached.
EISCONN	Indicates that the caller tried to connect while already connected.

Related reference:

"panic Kernel Service" on page 402

Related information:

Network Kernel Services

reconfig_register, reconfig_register_ext, reconfig_unregister, or reconfig_complete, reconfig_register_list Kernel Service

Purpose

Register and unregister reconfiguration handlers.

Syntax

```
#include <sys/dr.h>

int reconfig_register (handler, actions,
 h_arg, h_token, name)
int (*handler)(void *event, void *h_arg, int req,
void *resource_info);
int actions;
void *h_arg;
ulong *h_token;
char *name;

int reconfig_register_ext (handler, actions, h_arg, h_token, name)
int (*handler)(void *event, void *h_arg, unsigned long long req,
void *resource_info);
unsigned long long actions;
void *h_arg;
ulong *h_token;
char *name;

int reconfig_unregister (h_token)
ulong h_token;

void reconfig_complete (event, rc)
void *event;
int rc;

int reconfig_register_list (handler, event_list,  list_size, h_arg, h_token, name)
int (*handler)(void *event, void *h_arg, dr_kevent_t event_in_prog,
void *resource_info);
dr_kevent_t event_list[];
size_t list_size;
void *h_arg;
ulong *h_token;
char *name;
```

Description

The **reconfig_register, reconfig_register_ext**, **reconfig_register_list** and **reconfig_unregister** kernel services register and unregister reconfiguration handlers, which are invoked by the kernel both before and after DLPAR operations depending on the set of events specified by the kernel extension when registering.

Starting with AIX 6.1 with 6100-02, all future kernel extensions use the **reconfig_register_list** kernel service when registering for DLPAR operations. The **reconfig_register_list** kernel service supports previous and new DLPAR operations. The **reconfig_register** or **reconfig_register_ext** kernel services will no longer support all future DLPAR operations.

The **reconfig_complete** kernel service is used to indicate that the request has completed. If a kernel extension expects that the operation is likely to take a long time (several seconds), the handler must return **DR_WAIT** to the caller, but proceed with the request asynchronously. In this case, the handler must indicate that it has completed the request by invoking the **reconfig_complete** kernel service.

Parameters

Item	Description
actions	Allows the kernel extension to specify which of the following events require notification: • **DR_PMIG_CHECK** • **DR_PMIG_PRE** • **DR_PMIG_POST** • **DR_PMIG_POST_ERROR** • **DR_CAP_ADD_CHECK** • **DR_CAP_ADD_PRE** • **DR_CAP_ADD_POST** • **DR_CAP_ADD_POST_ERROR** • **DR_CAP_REMOVE_CHECK** • **DR_CAP_REMOVE_PRE** • **DR_CAP_REMOVE_POST** • **DR_CAP_REMOVE_POST_ERROR** • **DR_CPU_ADD_CHECK** • **DR_CPU_ADD_PRE** • **DR_CPU_ADD_POST** • **DR_CPU_ADD_POST_ERROR** • **DR_CPU_REMOVE_CHECK** • **DR_CPU_REMOVE_PRE** • **DR_CPU_REMOVE_POST** • **DR_CPU_REMOVE_POST_ERROR** • **DR_MEM_ADD_CHECK** • **DR_MEM_ADD_OP_POST** • **DR_MEM_ADD_PRE** • **DR_MEM_ADD_POST** • **DR_MEM_ADD_POST_ERROR** • **DR_MEM_REMOVE_CHECK** • **DR_MEM_REMOVE_OP_POST** • **DR_MEM_REMOVE_OP_PRE** • **DR_MEM_REMOVE_PRE** • **DR_MEM_REMOVE_POST** • **DR_MEM_REMOVE_POST_ERROR**
event	Passed to the handler and intended to be used only when calling the **reconfig_complete** kernel service.
event_list	Specifies which events require notification. For the supported values, see the **dr.h** file.
handler	Specifies the kernel extension function to be invoked.
h_arg	Specified by the kernel extension, remembered by the kernel along with the function descriptor for the handler, and passed to the handler when it is invoked. It is not used directly by the kernel, but is intended to support kernel extensions that manage multiple adapter instances. This parameter points to an adapter control block.
h_token	An output parameter that is used when unregistering the handler.
list_size	Specifies the memory size of the **event_list** array.

Item **Description**

name
Provided for information purposes and may be included within an error log entry, if the driver returns an error. It is provided by the kernel extension and must be limited to 15 ASCII characters.

rc
Can be set to **DR_FAIL** or **DR_SUCCESS**.

resource_info
Identifies the resource specific information for the current DLPAR request. If the request is cpu based, the *resource_info* data is provided through a **dri_cpu** structure. Otherwise a **dri_mem** structure is used. On a Micro-Partitioning partition, if the request is CPU-capacity based, the *resource_info* data is provided through a **dri_cpu_capacity** structure, which has the following format. The kernel extensions are not notified of changes in variable capacity weight in an uncapped Micro-Partitioning environment.

```
*/
struct dri_cpu_capacity {
 uint64_t ent_capacity; /* partition current entitled capacity*/
 int  delta_ent_cap; /* delta capacity added/removed*/
 int  status;  /* capacity update constrained or not */
};

/*
 * dri_cpu_capacity.status flags.
 */
#define CAP_UPDATE_SUCCESS 0x0
#define CAP_UPDATE_CONSTRAINED 0x1
```

Note: The capacity update is constrained by the Hypervisor.

If the request is memory capacity based, the *resource_info* data is provided through a **dri_mem_capacity** structure, which has the following format:

```
struct dri_mem_capacity {
  size64_t mem_capacity; /* partition current entitled capacity*/
  ssize64_t delta_mem_capacity;
  uint  flags;
  int  status;  /* capacity update constrained or not */
  uchar  reserved[7];
 };

  /*
    * dri_mem_capacity.status flags.
    */
  #define CAP_UPDATE_SUCCESS 0x0
  #define CAP_UPDATE_CONSTRAINED 0x1
```

Item	Description
req	Indicates the following DLPAR operation to be performed by the handler: • **DR_PMIG_CHECK** • **DR_PMIG_PRE** • **DR_PMIG_POST** • **DR_PMIG_POST_ERROR** • **DR_CAP_ADD_CHECK** • **DR_CAP_ADD_PRE** • **DR_CAP_ADD_POST** • **DR_CAP_ADD_POST_ERROR** • **DR_CAP_REMOVE_CHECK** • **DR_CAP_REMOVE_PRE** • **DR_CAP_REMOVE_POST** • **DR_CAP_REMOVE_POST_ERROR** • **DR_CPU_ADD_CHECK** • **DR_CPU_ADD_PRE** • **DR_CPU_ADD_POST** • **DR_CPU_ADD_POST_EEROR** • **DR_CPU_REMOVE_CHECK** • **DR_CPU_REMOVE_PRE** • **DR_CPU_REMOVE_POST** • **DR_CPU_REMOVE_POST_ERROR** • **DR_MEM_ADD_CHECK** • **DR_MEM_ADD_OP_POST** • **DR_MEM_ADD_PRE** • **DR_MEM_ADD_POST** • **DR_MEM_ADD_POST_ERROR** • **DR_MEM_REMOVE_CHECK** • **DR_MEM_REMOVE_OP_POST** • **DR_MEM_REMOVE_OP_PRE** • **DR_MEM_REMOVE_PRE** • **DR_MEM_REMOVE_POST** • **DR_MEM_REMOVE_POST_ERROR**

List of dr_kevent_t events

The following events are used with the **reconfig_register_list()** call for the **event_list** array:

- **DR_KEVENT_CPU_ADD_CHECK**
- **DR_KEVENT_CPU_ADD_PRE**
- **DR_KEVENT_CPU_ADD_POST**
- **DR_KEVENT_CPU_ADD_POST_ERROR**
- **DR_KEVENT_CPU_RM_CHECK**
- **DR_KEVENT_CPU_RM_PRE**
- **DR_KEVENT_CPU_RM_POST**
- **DR_KEVENT_CPU_RM_POST_ERROR**
- **DR_KEVENT_MEM_ADD_CHECK**
- **DR_KEVENT_MEM_ADD_PRE**
- **DR_KEVENT_MEM_ADD_POST**
- **DR_KEVENT_MEM_ADD_POST_ERROR**

- **DR_KEVENT_MEM_RM_CHECK**
- **DR_KEVENT_MEM_RM_PRE**
- **DR_KEVENT_MEM_RM_POST**
- **DR_KEVENT_MEM_RM_POST_ERROR**
- **DR_KEVENT_MEM_ADD_RES**
- **DR_KEVENT_MEM_RM_RES**
- **DR_KEVENT_CPU_CAP_ADD_CHECK**
- **DR_KEVENT_CPU_CAP_ADD_PRE**
- **DR_KEVENT_CPU_CAP_ADD_POST**
- **DR_KEVENT_CPU_CAP_ADD_POST_ERROR**
- **DR_KEVENT_CPU_CAP_RM_CHECK**
- **DR_KEVENT_CPU_CAP_RM_PRE**
- **DR_KEVENT_CPU_CAP_RM_POST**
- **DR_KEVENT_CPU_CAP_RM_POST_ERROR**
- **DR_KEVENT_MEM_RM_OP_PRE**
- **DR_KEVENT_MEM_RM_OP_POST**
- **DR_KEVENT_MEM_ADD_OP_POST**
- **DR_KEVENT_PMIG_CHECK**
- **DR_KEVENT_PMIG_PRE**
- **DR_KEVENT_PMIG_POST**
- **DR_KEVENT_PMIG_POST_ERROR**
- **DR_KEVENT_PMIG_POST_INTERNAL**
- **DR_KEVENT_WMIG_CHECK**
- **DR_KEVENT_WMIG_PRE**
- **DR_KEVENT_WMIG_POST**
- **DR_KEVENT_WMIG_POST_ERROR**
- **DR_KEVENT_WMIG_CHECKPOINT_CHECK**
- **DR_KEVENT_WMIG_CHECKPOINT_PRE**
- **DR_KEVENT_WMIG_CHECKPOINT_DOIT**
- **DR_KEVENT_WMIG_CHECKPOINT_ERROR**
- **DR_KEVENT_WMIG_CHECKPOINT_POST**
- **DR_KEVENT_WMIG_CHECKPOINT_POST_ERROR**
- **DR_KEVENT_WMIG_RESTART_CHECK**
- **DR_KEVENT_WMIG_RESTART_PRE**
- **DR_KEVENT_WMIG_RESTART_DOIT**
- **DR_KEVENT_WMIG_RESTART_ERROR**
- **DR_KEVENT_WMIG_RESTART_POST**
- **DR_KEVENT_WMIG_RESTART_POST_ERROR**
- **DR_KEVENT_MEM_CAP_ADD_CHECK**
- **DR_KEVENT_MEM_CAP_ADD_PRE**
- **DR_KEVENT_MEM_CAP_ADD_POST**
- **DR_KEVENT_MEM_CAP_ADD_POST_ERROR**
- **DR_KEVENT_MEM_CAP_RM_CHECK**
- **DR_KEVENT_MEM_CAP_RM_PRE**
- **DR_KEVENT_MEM_CAP_RM_POST**

- **DR_KEVENT_MEM_CAP_RM_POST_ERROR**
- **DR_KEVENT_MEM_CAP_WGT_ADD_CHECK**
- **DR_KEVENT_MEM_CAP_WGT_ADD_PRE**
- **DR_KEVENT_MEM_CAP_WGT_ADD_POST**
- **DR_KEVENT_MEM_CAP_WGT_ADD_POST_ERROR**
- **DR_KEVENT_MEM_CAP_WGT_RM_CHECK**
- **DR_KEVENT_MEM_CAP_WGT_RM_PRE**
- **DR_KEVENT_MEM_CAP_WGT_RM_POST**
- **DR_KEVENT_MEM_CAP_WGT_RM_POST_ERROR**
- **DR_KEVENT_TOPOLOGY_PRE**
- **DR_KEVENT_TOPOLOGY_POST**
- **DR_KEVENT_AME_FACTOR_CHECK**
- **DR_KEVENT_AME_FACTOR_PRE**
- **DR_KEVENT_AME_FACTOR_POST**
- **DR_KEVENT_AME_FACTOR_POST_ERROR**

Return Values

Upon successful completion, the **reconfig_register, reconfig_register_ext** and **reconfig_unregister** kernel services return zero. If unsuccessful, the appropriate **errno** value is returned.

Execution Environment

The **reconfig_register, reconfig_register_ext**, **reconfig_unregister**, and **handler** interfaces are invoked in the process environment only.

The **reconfig_complete** kernel service may be invoked in the process or interrupt environment.

Related information:

Making Kernel Extensions DLPAR-Aware

refmon Kernel Service

Purpose

Performs various access checks such as privileges, authorizations, discretionary access control checks and so on.

Syntax

```
#include <refmon.h>

int refmon (crp, action, flags, nargs, args[])
cred_t *crp;
rfm_action_t action;
uint_t flags;
int nargs;
void *args[];
```

Parameters

Item	Description
crp	Specifies the caller's (subject) credentials; If NULL, then current process credentials are referenced.
action	Specifies the type of access check that needs to be carried out.
flags	Enables auditing of this event. You can only set this parameter to the value of REFMON_AUDIT.
nargs	Specifies the number of arguments in the *args* parameter.
args	Specifies an array of void pointers used as input to the **refmon** kernel service based on the *action* parameter.

Description

The **refmon** kernel service provides an interface to perform various access checks. You can call the **refmon** kernel service to determine access to system resources. Most of the actions that are passed to the **refmon** kernel service check for specific privileges. Many of the system calls and kernel services call the **refmon** kernel service to check whether you are authorized or privileged to use such functions. The *action* parameter determines which type of checks needs to be performed. The **sys/refmon.h** header file contains a complete list of these actions and their corresponding description.

Execution Environment

The **refmon** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Success.
EINVAL	The *action* parameter is not valid or a value that is not allowed is passed in for an action.
EPERM	The caller does not have permission to perform the intended action.

Related information:
Security Kernel Services

register_HA_handler Kernel Service

Purpose

Registers a High Availability Event Handler with the Kernel.

Syntax

```
#include <sys/high_avail.h>

int register_HA_handler (ha_handler)
ha_handler_ext_t * ha_handler;
```

Parameter

Item	Description
ha_handler	Specifies a pointer to a structure of the type **ha_handler_ext_t** as defined in **/usr/include/sys/high_avail.h**.

Description

The **register_HA_handler** kernel registers the **High Availability Event Handler (HAEH)** function to those kernel extensions that need to be made aware of high availability events such as processor deallocation. This function is called by the kernel, at base level, when a high availability event is initiated, due to some hardware fault.

The **ha_handler_ext_t** structure has 3 fields:

Field	Description
_fun	Contains a pointer to the high availability event handler function.
_data	Contains a user defined value which will be passed as an argument by the kernel when calling the function.
_name	Component name

When a high availability event is initiated, the kernel calls *_fun()* at base level (that is, process environment) with 2 parameters:

- The first is the data the user passed in the *_data* field at registration time.
- The second is a pointer to a **haeh_event_t** structure defined in **/usr/include/sys/high_avail.h**.

The fields of interest in this structure are:

Field	Description
_magic	Identifies the event type. The only possible value is **HA_CPU_FAIL**.
dealloc_cpu	The logical number of the CPU being deallocated.

The high availability even handler, in addition to user specific functions, must unbind its threads bound to *dealloc_cpu* and stop the timer request blocks (TRB) started by those bound threads when applicable.

The high availability event handler must return one of the following values:

Value	Description
HA_ACCEPTED	The user processing of the event has succeeded.
HA_REFUSED	The user processing of the event was not successful.

Any return value different from **HA_ACCEPTED** causes the kernel to abort the processing of the event. In the case of a processor failure, the processor deallocation is aborted. In this case, a `CPU_DEALLOC_ABORTED` error log entry is created, and the value passed in the *_name* field appears in the detailed data area of the error log entry.

An extension may register the same HAEH *N* times (*N* > 1). Although it is considered as an incorrect behaviour, no error is reported. The given HAEH is invoked *N* times for each HA event. This handler has to be unregistered as many times as it was registered.

Since the kernel calls the HAEH in turn, it is possible for a HAEH to be called multiple times for the same event. The kernel extensions should be ready to deal with this possibility. For example, two kernel extensions **K1** and **K2** have registered HA Handlers. A CPU deallocation is initiated. The HAEH for **K1** gets invoked, does its job and returns `HA_ACCEPTED`. **K2** gets invoked next and for some reason returns `HA_REFUSED`. The deallocation is aborted, and an error log entry reports **K2** as the reason for failure. Later, the system administer unloads **K2** and restarts the deallocation by manually running **ha_star**. The result is that the HAEH for **K1** gets invoked again with the same parameters.

Execution Environment

The **register_HA_handler** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.

A non zero value indicates an error.

Related reference:

"unregister_HA_handler Kernel Service" on page 521

Related information:

RAS Kernel Services

rmalloc Kernel Service

Purpose

Allocates an area of memory from the **real_heap** heap.

Syntax

```
#include <sys/types.h>

caddr_t rmalloc (size, align)
int size
int align
```

Parameters

Item	Description
size	Specifies the number of bytes to allocate.
align	Specifies alignment characteristics.

Description

The **rmalloc** kernel service allocates an area of memory from the contiguous real memory heap. This area is the number of bytes in length specified by the *size* parameter and is aligned on the byte boundary specified by the *align* parameter. The *align* parameter is actually the log base 2 of the desired address boundary. For example, an *align* value of 4 requests that the allocated area be aligned on a 16-byte boundary.

The contiguous real memory heap, **real_heap**, is a heap of contiguous real memory pages located in the low 16MB of real memory. This heap is virtually mapped into the kernel extension's address space. By nature, this heap is implicitly pinned, so no explicit pinning of allocated regions is necessary.

The **real_heap** heap is useful for devices that require DMA transfers greater than 4K but do not provide a scatter/gather capability. Such a device must be given contiguous bus addresses by its device driver. The device driver should pass the **DMA_CONTIGUOUS** flag on its **d_map_init** call in order to obtain contiguous mappings. On certain platforms it is possible that a **d_map_init** call using the **DMA_CONTIGUOUS** flag could fail. In this case, the device driver can make use of the **real_heap** heap (using **rmalloc**) to obtain contiguous bus addresses for its device driver. Because the **real_heap** heap is a limited resource, device drivers should always attempt to use the **DMA_CONTIGUOUS** flag first.

On unsupported platforms, the **rmalloc** service returns NULL if the requested memory cannot be allocated.

The **rmfree** kernel service should be called to free allocation from a previous **rmalloc** call. The **rmalloc** kernel service can be called from the process environment only.

Return Values

Upon successful completion, the **rmalloc** kernel service returns the address of the allocated area. A **NULL** pointer is returned if the requested memory cannot be allocated.

Related reference:

"rmfree Kernel Service"

rmfree Kernel Service

Purpose

Frees memory allocated by the **rmalloc** kernel service.

Syntax

```
#include <sys/types.h>

int rmfree ( pointer,  size)
caddr_t pointer
int size
```

Parameters

Item	Description
pointer	Specifies the address of the area in memory to free.
size	Specifies the size of the area in memory to free.

Description

The **rmfree** kernel service frees the area of memory pointed to by the *pointer* parameter in the contiguous real memory heap. This area of memory must be allocated with the **rmalloc** kernel service, and the *pointer* must be the pointer returned from the corresponding **rmalloc** kernel service call. Also, the *size* must be the same size that was used on the corresponding **rmalloc** call.

Any memory allocated in a prior **rmalloc** call must be explicitly freed with an **rmfree** call. This service can be called from the process environment only.

Return Values

Item	Description
0	Indicates successful completion.
-1	Indicates one of the following: • The area was not allocated by the **rmalloc** kernel service. • The heap was not initialized for memory allocation.

Related reference:

"rmalloc Kernel Service" on page 448

rmmap_create Kernel Service

Purpose

Defines an Effective Address [EA] to Real Address [RA] translation region.

Syntax

```
#include <sys/ioacc.h>
#include <sys/adspace.h>
```

```
int rmmap_create ( eaddrp,  iomp,  flags)
void **eaddrp;
struct io_map *iomp;
int flags;
```

Parameters

Item	Description
eaddr	Required process effective address of the mapping region.
iomp	The bus memory to which the effective address described by the *eaddr* parameter must correspond. For real memory, the bus id must be set to **REALMEM_BID** and the bus address must be set to the real memory address. The size field must be at least **PAGESIZE**, no larger than **SEGSIZE**, and a multiple of **PAGESIZE**. The key must be set to **IO_MEM_MAP**. The flags field is not used.
flags	The flags select page and segment attributes of the translation. Not all page attribute flags are compatible. The valid combinations of page attribute flags follow.

RMMAP_PAGE_W
PowerPC "Write Through" page attribute. Write-through mode is not supported, and if this flag is set, **EINVAL** is reported.

RMMAP_PAGE_I
PowerPC "Cache Inhibited" page attribute. This flag is valid for I/O mappings, but is not allowed for real memory mappings.

RMMAP_PAGE_M
PowerPC "Memory Coherency Required" page attribute. This flag is optional for I/O mappings; however, it is required for memory mappings. The default operating mode for real memory pages has this bit set.

RMMAP_PAGE_G
PowerPC "Guarded" page attribute. This flag is optional for I/O mappings, and must be 0 for real memory mappings. Although optional for I/O, it is recommended that this flag must be set for I/O mappings. When set, the processor does not make unnecessary (speculative) references to the page. It includes out of order read or write operations and branch fetching. When clear, normal PowerPC speculative execution rules apply.

RMMAP_RDONLY
When set, the page protection bits used in the **HTAB** does not allow write operations regardless of the setting of the key bit in the associated segment register. Exactly one of **RMMAP_RDONLY** and **RMMAP_RDWR** must be specified.

RMMAP_RDWR
When set, the page protection bits used in the **HTAB** allows read and write operations regardless of the setting of the key bit in the associated segment register. Exactly one of: **RMMAP_RDONLY**, and **RMMAP_RDWR** must be specified.

RMMAP_PRELOAD
When set, the protection attributes of this region are entered immediately into the hardware page table. It is very slow initially, but prevents each referenced page in the region from faulting in separately. It is only advisory. This flag is not maintained as an attribute of the map region, it is used only during the current call.

RMMAP_INHERIT
When set, this protection attribute specifies that the translation region created by this **rmmap_create** invocation must be inherited on a **fork** operation, to the child process. This inheritance is achieved with copy-semantics. The child has its own private mapping to the same I/O or real memory address range as the parent.

Description

The translation regions that are created with **rmmap_create** kernel service are maintained in I/O mapping segments. Any single such segment might translate up to 256 Megabytes of real memory or memory mapped I/O in a single region. The only granularity for which the **rmmap_remove** service might be started is a single mapping that is created by a single call to the **rmmap_create**.

There are constraints on the size of the mapping and the *flags* parameter, described later, which causes the call to fail regardless of whether adequate effective address space exists.

If **rmmap_create** kernel service is called with the effective address of zero, the function attempts to find free space in the process address space. If successful, an I/O mapping segment is created and the effective address (which is passed by reference) is changed to the effective address which is mapped to the first page of the *iomp* memory.

If **rmmap_create** kernel service is called with a non-zero effective address, it is taken as the required effective address which must translate to the passed *iomp* memory. This function verifies that the requested range is free. If not, it fails and returns **EINVAL**. If the mapping at the effective address is not contained in a single segment, the function fails and returns **ENOSPC**. Otherwise, the region is allocated and the effective address is not modified. The effective address is mapped to the first page of the *iomp* memory. References outside of the mapped regions but within the same segment are invalid.

The effective address (if provided) and the bus address must be a multiple of **PAGESIZE** or **EINVAL** is returned.

I/O mapping segments are not inherited by child processes after a **fork** subroutine.

I/O mapping segments are not inherited by child processes after a **fork** subroutine, except when **RMMAP_INHERIT** is specified. These segments are deleted by **exec**, **exit**, or **rmmap_remove** of the last range in a segment.

Only certain combinations of flags are permitted, depending on the type of memory that is mapped. For real memory mappings, **RMMAP_PAGE_M** is required while **RMMAP_PAGE_W**, **RMMAP_PAGE_I**, and **RMMAP_PAGE_G** are not allowed. For I/O mappings, it is valid to specify only **RMMAP_PAGE_M**, with no other page attribute flags. It is also valid to specify **RMMAP_PAGE_I** and optionally, either or both of **RMMAP_PAGE_M**, and **RMMAP_PAGE_G**. **RMMAP_PAGE_W** is never allowed.

The real address range that is described by the *iomp* parameter must be unique within this I/O mapping segment.

Execution Environment

The **rmmap_create** kernel service can be called only from the process environment.

Return Values

On successful completion, **rmmap_create** kernel service returns zero and modifies the effective address to the value at which the newly created mapping region was attached to the process address space. Otherwise, it returns one of following errors:

Item	Description
EINVAL	Some type of parameter error occurred. These parameters include, but are not limited to, size errors and mutually exclusive flag selections.
ENOMEM	The operating system cannot allocate the necessary data structures to represent the mapping.
ENOSPC	Effective address space exhausted in the region indicated by *eaddr*.
EPERM	This hardware platform does not implement this service.

Implementation Specifics

This service only functions on PowerPC microprocessors.

Related reference:

"rmmap_remove Kernel Service" on page 453

Related information:

Memory Kernel Services

rmmap_getwimg Kernel Service

Purpose

Returns wimg information about a particular effective address range within an effective address to real address translation region.

Syntax

```
#include <sys/adspace.h>

int rmmap_getwimg(eaddr, npages, results)
unsigned long long eaddr;
unsigned int  npages;
char* results;
```

Parameters

Item	Description
eaddr	The process effective address of the start of the desired mapping region. This address should point somewhere inside the first page of the range. This address is interpreted as a 64-bit quantity if the current user address space is 64-bits, and is interpreted as a 32-bit (not remapped) quantity if the current user address space is 32-bits.
npages	The number of pages whose wimg information is returned, starting from the page indicated by **eaddr**.
results	This is an array of bytes, where the wimg information is returned. The address of this is passed in by the caller, and **rmmap_getwimg** stores the wimg information for each page in the range in each successive byte in this array. The size of this array is indicated by *npages* as specified by the caller. The caller is responsible for ensuring that the storage allocated for this array is large enough to hold *npage* bytes.

Description

The wimg information corresponding to the input effective address range is returned.

This routine only works for regions previously mapped with an I/O mapping segment as created by **rmmap_create**.

npages should not be such that the range crosses a segment boundary. If it does, EINVAL is returned.

The wimg information is returned in the **results** array. Each element of the **results** array is a character. Each character may be added with the following fields to examine wimg information: **RMMAP_PAGE_W**, **RMMAP_PAGE_I**, **RMMAP_PAGE_M** or **RMMAP_PAGE_G**. The array is valid if the return value is 0.

Execution Environment

The **rmmap_getwimg** kernel service is called from the process environment only.

Return Values

Item	Description
0	Successful completion. Indicates that the *results* array is valid and should be examined.
EINVAL	An error occurred. Most likely the region was not mapped via **rmmap_create** previously..
EINVAL	Input range crosses a certain boundary.
EINVAL	The hardware platform does not implement this service.

Implementation Specifics

This service only functions on PowerPC microprocessors.

Related reference:

"rmmap_create Kernel Service" on page 449

"rmmap_remove Kernel Service"

rmmap_remove Kernel Service

Purpose

Destroys an effective address to real address translation region.

Syntax

```
#include <sys/adspace.h>
int rmmap_remove (eaddrp);
void **eaddrp;
```

Parameters

Item	Description
eaddrp	Pointer to the process effective address of the desired mapping region.

Description

Destroys an effective address to real address translation region. If **rmmap_remove** kernel service is called with the effective address within the region of a previously created I/O mapping segment, the region is destroyed. This service must be called from the process level.

Execution Environment

The **rmmap_remove** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL	The provided *eaddr* does not correspond to a valid I/O mapping segment.
EINVAL	This hardware platform does not implement this service.

Implementation Specifics

This service only functions on PowerPC microprocessors.

Related reference:

"rmmap_create Kernel Service" on page 449

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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

Item	Description
ro	Specifies the route.

Description

The **rtalloc** kernel service allocates a route, which consists of a destination address and a reference to a routing entry.

Execution Environment

The **rtalloc** kernel service can be called from either the process or interrupt environment.

Return Values

The **rtalloc** service has no return values.

Example

To allocate a route, invoke the **rtalloc** kernel service as follows:

```
rtalloc(ro);
```

Related information:

Network Kernel Services

rtalloc_gr Kernel Service

Purpose

Allocates a route.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <net/route.h>

void rtalloc_gr ( ro,  gidlist)
register struct route *ro;
struct gidstruct *gidlist;
```

Parameter

Item	Description
ro	Specifies the route.
gidlist	Points to the group list.

Description

The **rtalloc_gr** kernel service allocates a route, which consists of a destination address and a reference to a routing entry.

A route can be allocated only if its group id restrictions specify that it can be used by a user with the *gidlist* that is passed in.

Execution Environment

The **rtalloc_gr** kernel service can be called from either the process or interrupt environment.

Return Values

The **rtalloc_gr** service has no return values.

Example

To allocate a route, invoke the **rtalloc_gr** kernel service as follows:

```
rtalloc_gr (ro, gidlist);
```

Related reference:

"rtalloc Kernel Service" on page 454

Related information:

Network Kernel Services

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

Item	Description
rt	Specifies the routing table entry.

Description

The **rtfree** kernel service frees the entry it is passed from the routing table. If the route does not exist, the **panic** service is called. Otherwise, the **rtfree** service frees the **mbuf** structure that contains the route and decrements the routing reference counters.

Execution Environment

The **rtfree** kernel service can be called from either the process or interrupt environment.

Return Values

The **rtfree** kernel service has no return values.

Example

To free a routing table entry, invoke the **rtfree** kernel service as follows:

```
rtfree(rt);
```

Related reference:

"panic Kernel Service" on page 402

Related information:

Network Kernel Services

rtinit Kernel Service

Purpose

Sets up a routing table entry typically for a network interface.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/socket.h>
#include <net/route.h>

int rtinit (ifa, cmd, flags)
struct ifaddr * ifa;
int  cmd,  flags;
```

Parameters

Item	Description
ifa	Specifies the address of an **ifaddr** structure containing destination address, interface address, and netmask.
cmd	Specifies a request to add or delete route entry.
flags	Identifies routing flags, as defined in the **/usr/include/net/route.h** file.

Description

The **rtinit** kernel service creates a routing table entry for an interface. It builds an **rtentry** structure using the values in the *ifa* and *flags* parameters.

The **rtinit** service then calls the **rtrequest** kernel service and passes the *cmd* parameter and the **rtentry** structure to process the request. The *cmd* parameter contains either the value **RTM_ADD** (a request to add the route entry) or the value **RTM_DELETE** (delete the route entry). Valid routing flags to set are defined in the **/usr/include/route.h** file.

Execution Environment

The **rtinit** kernel service can be called from either the process or interrupt environment.

Return Values

The **rtinit** kernel service returns values from the **rtrequest** kernel service.

Example

To set up a routing table entry, invoke the **rtinit** kernel service as follows:

```
rtinit(ifa, RMT_ADD, flags ( RTF_DYNAMIC);
```

Related reference:

"rtrequest Kernel Service" on page 458

Related information:

Network Kernel Services

rtredirect Kernel Service

Purpose

Forces a routing table entry with the specified destination to go through a given gateway.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>
#include <net/route.h>

rtredirect ( dst, gateway, netmask, flags, src, rtp)
struct sockaddr *dst, *gateway, *netmask, *src;
int flags;
struct rtentry **rtp;
```

Parameters

Item	Description
dst	Specifies the destination address.
gateway	Specifies the gateway address.
netmask	Specifies the network mask for the route.
flags	Indicates routing flags as defined in the **/usr/include/net/route.h** file.
src	Identifies the source of the redirect request.
rtp	Indicates the address of a pointer to a **rtentry** structure. Used to return a constructed route.

Description

The **rtredirect** kernel service forces a routing table entry for a specified destination to go through the given gateway. Typically, the **rtredirect** service is called as a result of a routing redirect message from the network layer. The *dst*, *gateway*, and *flags* parameters are passed to the **rtrequest** kernel service to process the request.

Execution Environment

The **rtredirect** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
0	Indicates a successful operation.

If a bad redirect request is received, the routing statistics counter for bad redirects is incremented.

Example

To force a routing table entry with the specified destination to go through the given gateway, invoke the **rtredirect** kernel service:

```
rtredirect(dst, gateway, netmask, flags, src, rtp);
```

Related reference:

"rtinit Kernel Service" on page 456

Related information:

Network Kernel Services

rtrequest Kernel Service

Purpose

Carries out a request to change the routing table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/af.h>
#include <net/route.h>

int rtrequest ( req,  dst,  gateway,  netmask,  flags,  ret_nrt)
int req;
struct sockaddr *dst, *gateway, *netmask;
int flags;
struct rtentry **ret_nrt;
```

Parameters

Item	Description
req	Specifies a request to add or delete a route.
dst	Specifies the destination part of the route.
gateway	Specifies the gateway part of the route.
netmask	Specifies the network mask to apply to the route.
flags	Identifies routing flags, as defined in the **/usr/include/net/route.h** file.
ret_nrt	Specifies to return the resultant route.

Description

The **rtrequest** kernel service carries out a request to change the routing table. Interfaces call the **rtrequest** service at boot time to make their local routes known for routing table ioctl operations. Interfaces also call the **rtrequest** service as the result of routing redirects. The request is either to add (if the *req* parameter has a value of **RMT_ADD**) or delete (the *req* parameter is a value of **RMT_DELETE**) the route.

Execution Environment

The **rtrequest** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
0	Indicates a successful operation.
ESRCH	Indicates that the route was not there to delete.
EEXIST	Indicates that the entry the **rtrequest** service tried to add already exists.
ENETUNREACH	Indicates that the **rtrequest** service cannot find the interface for the route.
ENOBUFS	Indicates that the **rtrequest** service cannot get an **mbuf** structure to add an entry.

Example

To carry out a request to change the routing table, invoke the **rtrequest** kernel service as follows:

```
rtrequest(RTM_ADD, dst, gateway, netmask, flags, &rtp);
```

Related reference:

"rtinit Kernel Service" on page 456

Related information:

Network Kernel Services

rtrequest_gr Kernel Service

Purpose

Carries out a request to change the routing table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/af.h>
#include <net/route.h>

int rtrequest_gr ( req,  dst,  gateway,  netmask,  flags,  ret_nrt, rt_parm)
int req;
struct sockaddr *dst, *
gateway
, *netmask;
int flags;
struct rtentry **
ret_nrt;
struct rtreq_parm *
rt_parm;
```

Parameters

Item	Description
req	Specifies a request to add or delete a route.
dst	Specifies the destination part of the route.
gateway	Specifies the gateway part of the route.
netmask	Specifies the network mask to apply to the route.
flags	Identifies routing flags, as defined in the **/usr/include/net/route.h** file.
ret_nrt	Specifies to return the resultant route.
rt_parm	Points to the **rtreq_parm** structure. The **/usr/include/net/radix.h** file contains the **rtreq_parm** structure. Through this structure, the route attributes like group list, policy, weight, WPAR ID, interface can be specified.

Description

The **rtrequest_gr** kernel service carries out a request to change the routing table. Interfaces call the **rtrequest_gr** service at boot time to make their local routes known for routing table ioctl operations. Interfaces also call the **rtrequest_gr** service as the result of routing redirects. The request is either to add (if the *req* parameter has a value of **RMT_ADD**) or delete (the *req* parameter is a value of **RMT_DELETE**) the route.

Execution Environment

The **rtrequest_gr** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
0	Indicates a successful operation.
ESRCH	Indicates that the route was not there to delete.
EEXIST	Indicates that the entry the **rtrequest_gr** service tried to add already exists.
ENETUNREACH	Indicates that the **rtrequest_gr** service cannot find the interface for the route.
ENOBUFS	Indicates that the **rtrequest_gr** service cannot get an **mbuf** structure to add an entry.

Example

To carry out a request to change the routing table, invoke the **rtrequest_gr** kernel service as follows:

```
rtrequest_gr(RTM_ADD, dst, gateway, netmask, flags, &rtp, &rtreq);
```

Related reference:

"rtinit Kernel Service" on page 456

"rtrequest Kernel Service" on page 458

Related information:

Network Kernel Services

rusage_incr Kernel Service

Purpose

Increments a field of the **rusage** structure.

Syntax

```
#include <sys/encap.h>

void rusage_incr ( field,  amount)
int field;
int amount;
```

Parameters

Item | **Description**
field | Specifies the field to increment. It must have one of the following values:

RUSAGE_INBLOCK
Denotes the `ru_inblock` field. This field specifies the number of times the file system performed input.

RUSAGE_OUTBLOCK
Denotes the `ru_outblock` field. This field specifies the number of times the file system performed output.

RUSAGE_MSGRCV
Denotes the `ru_msgrcv` field. This field specifies the number of IPC messages received.

RUSAGE_MSGSENT
Denotes the `ru_msgsnd` field. This field specifies the number of IPC messages sent.

amount | Specifies the amount to increment to the field.

Description

The **rusage_incr** kernel service increments the field specified by the *field* parameter of the calling process' **rusage** structure by the amount *amount*.

Execution Environment

The **rusage_incr** kernel service can be called from the process environment only.

Return Values

The **rusage_incr** kernel service has no return values.

Related information:

getrusage subroutine

Process and Exception Management Kernel Services

s

The following kernel services begin with the with the letter s.

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

Item	Description
anisr	Specifies the software interrupt number to issue. Refer to **netisr.h** for the range of values of *anisr* that are already in use. Also, other kernel extensions that are not AIX and that use network ISRs currently running on the system can make use of additional values not mentioned in **netisr.h**.

Description

The **schednetisr** kernel service schedules or calls a network interrupt service routine. The **add_netisr** kernel service establishes interrupt service routines. If the service was added with a service level of **NET_OFF_LEVEL**, the **schednetisr** kernel service directly calls the interrupt service routine. If the service level was **NET_KPROC**, a network kernel dispatcher is notified to call the interrupt service routine.

Execution Environment

The **schednetisr** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
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.

Related reference:

"add_netisr Kernel Service" on page 10

"del_netisr Kernel Service" on page 65

Related information:

Network Kernel Services

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

Item	Description
id	Indicates a primary resource identification value. This value along with the subidentifier (specified by the *subid* parameter) is used by the kernel to notify the appropriate processes of the occurrence of the indicated events. If the resource on which the event has occurred is a device driver, this parameter must be the device major/minor number (that is, a **dev_t** structure that has been cast to an **int**). The kernel has reserved values for the *id* parameter that do not conflict with possible device major or minor numbers for sockets, message queues, and named pipes.
subid	Helps identify the resource on which the event has occurred for the kernel. For a multiplexed device driver, this is the number of the channel on which the requested events occurred. If the device driver is nonmultiplexed, the *subid* parameter must be set to 0.
rtnevents	Consists of a set of bits indicating the requested events that have occurred on the specified device or channel. These flags have the same definition as the event flags that were provided by the *events* parameter on the unsatisfied call to the object's select routine.

Description

The **selnotify** kernel service should be used by device drivers that support select or poll operations. It is also used by the kernel to support select or poll requests to sockets, named pipes, and message queues.

The **selnotify** kernel service wakes up processes waiting on a **select** or **poll** subroutine. The processes to be awakened are those specifying the given device and one or more of the events that have occurred on the specified device. The **select** and **poll** subroutines allow a process to request information about one or more events on a particular device. If none of the requested events have yet happened, the process is put to sleep and re-awakened later when the events actually happen.

The **selnotify** service should be called whenever a previous call to the device driver's **ddselect** entry point returns and both of the following conditions apply:

- The status of all requested events is false.
- Asynchronous notification of the events is requested.

The **selnotify** service can be called for other than these conditions but performs no operation.

Sequence of Events for Asynchronous Notification

The device driver must store information about the events requested while in the driver's **ddselect** routine under the following conditions:

- None of the requested events are true (at the time of the call).
- The **POLLSYNC** flag is not set in the *events* parameter.

The **POLLSYNC** flag, when not set, indicates that asynchronous notification is desired. In this case, the **selnotify** service should be called when one or more of the requested events later becomes true for that device and channel.

When the device driver finds that it can satisfy a **select** request, (perhaps due to new input data) and an unsatisfied request for that event is still pending, the **selnotify** service is called with the following items:

- Device major and minor number specified by the *id* parameter
- Channel number specified by the *subid* parameter
- Occurred events specified by the *rtnevents* parameter

These parameters describe the device instance and requested events that have occurred on that device. The notifying device driver then resets its requested-events flags for the events that have occurred for that device and channel. The reset flags thus indicate that those events are no longer requested.

If the *rtnevents* parameter indicated by the call to the **selnotify** service is no longer being waited on, no processes are awakened.

Execution Environment

The **selnotify** kernel service can be called from either the process or interrupt environment.

Return Values

The **selnotify** service has no return values.

Implementation Specifics

The **selnotify** kernel service is part of Base Operating System (BOS) Runtime.

Related reference:

“ddselect Device Driver Entry Point” on page 632

“fp_poll Kernel Service” on page 160

“fp_select Kernel Service” on page 166

“selreg Kernel Service”

Related information:

poll subroutine

select subroutine

Kernel Extension and Device Driver Management Kernel Services

selreg Kernel Service

Purpose

Registers an asynchronous poll or select request with the kernel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/poll.h>

int selreg ( corl,  dev_id,  unique_id,  reqevents,  notify)
int corl;
int dev_id;
int unique_id;
ushort reqevents;
void (*notify) ( );
```

Parameters

Item	Description
corl	The correlator for the poll or select request. The *corl* parameter is used by the **poll** and **select** subroutines to correlate the returned events in a specific select control block with a process' file descriptor or message queue.
dev_id	Primary resource identification value. Along with the *unique_id* parameter, the *dev_id* parameter is used to record in the select control block the resource on which the requested poll or select events are expected to occur.
unique_id	Unique resource identification value. Along with the *dev_id* parameter, the *unique_id* parameter denotes the resource on which the requested events are expected to occur. For a multiplexed device driver, this parameter specifies the number of the channel on which the requested events are expected to occur. For a nonmultiplexed device driver, this parameter must be set to 0.
reqevents	Requested events parameter. The *reqevents* parameter consists of a set of bit flags denoting the events for which notification is being requested. These flags have the same definitions as the event flags provided by the *events* parameter on the unsatisfied call to the object's **select** subroutine (see the **sys/poll.h** file for the definitions). **Note:** The **POLLSYNC** bit flag should not be set in this parameter.
notify	Notification routine entry point. This parameter points to a notification routine used for nested poll and select calls.

Description

The **selreg** kernel service is used by **select** file operations in the top half of the kernel to register an unsatisfied asynchronous poll or select event request with the kernel. This registration enables later calls to the **selnotify** kernel service from resources in the bottom half of the kernel to correctly identify processes awaiting events on those resources.

The event requests may originate from calls to the **poll** or **select** subroutine, from processes, or from calls to the **fp_poll** or **fp_select** kernel service. A **select** file operation calls the **selreg** kernel service under the following circumstances:

- The poll or select request is asynchronous (the **POLLSYNC** flag is not set for the requested event's bit flags).
- The poll or select request determines (by calling the underlying resource's **ddselect** entry point) that the requested events have not yet occurred.

A registered event request takes the form of a select control block. The select control block is a structure containing the following:

- Requested event bit flags
- Returned event bit flags
- Primary resource identifier
- Unique resource identifier
- Pointer to a **proc** table entry
- File descriptor correlator
- Pointer to a notification routine that is non-null only for nested calls to the **poll** and **select** subroutines

The **selreg** kernel service allocates and initializes a select control block each time it is called.

When an event occurs on a resource that supports the **select** file operation, the resource calls the **selnotify** kernel service. The **selnotify** kernel service locates all select control blocks whose primary and unique identifiers match those of the resource, and whose requested event flags match the occurred events on the resource. Then, for each of the matching control blocks, the **selnotify** kernel service takes one of two courses of action, depending upon whether the control block's notification routine pointer is non-null (nested) or null (non-nested):

- In nested calls to the **select** or **poll** subroutines, the notification routine is called with the primary and unique resource identifiers, the returned event bit flags, and the process identifiers.
- In non-nested calls to the **select** or **poll** subroutine (the usual case), the SSEL bit of the process identified in the block is cleared, the returned event bit flags in the block are updated, and the process is awakened. A process awakened in this manner completes the **poll** or **select** call in which it was sleeping. The **poll** or **select** subroutine then collects the returned event bit flags in its processes' select control blocks for return to the user mode process, deallocates the control blocks, and returns tallys of the numbers of requested events that occurred to the user process.

Execution Environment

The **selreg** kernel service can be called from the process environment only.

Returns Values

Item	Description
0	Indicates successful completion.
EAGAIN	Indicates the **selreg** kernel service was unable to allocate a select control block.

Related reference:

“ddselect Device Driver Entry Point” on page 632

“fp_select Kernel Service” on page 166

Related information:

select subroutine

Kernel Extension and Device Driver Management Kernel Services

set_pag or set_pag64 Kernel Service

Purpose

Sets a Process Authentication Group (PAG) value for the current process.

Syntax

```
#include <sys/cred.h>

int set_pag ( type, pag )
int type;
int pag;

int set_pag64 ( type, pag )
int type;
uint64_t *pag;
```

Parameters

Item	Description
type	PAG type to change
pag	PAG value

Description

The **set_pag** or **set_pag64** kernel service copies the requested PAG for the current process. The caller must synchronize the **set_pag** and **set_pag64** kernel services with **validate_pag** because **set_pag** and **set_pag64** do not lock process creation across the system. The value of *type* must be a defined PAG ID. The PAG ID for the Distributed Computing Environment (DCE) is 0.

Execution Environment

The **set_pag** and **set_pag64** kernel services can be called from the process environment only.

Return Values

A value of `0` is returned upon successful completion. Upon failure, a `-1` is returned and **errno** is set to a value that explains the error.

Error Codes

The **set_pag** and **set_pag64** kernel services fails if one or both of the following conditions are true:

Item	Description
EINVAL	Invalid PAG specification

Related information:

Security Kernel Services

setioctlrv Subroutine

Purpose

Sets a value to be returned by an **ioctl** routine.

Syntax

```
void setioctlrv (ioctlrv)
int ioctlrv;
```

Parameters

Item	Description
ioctlrv	Specifies an integer value to be returned by a successful completion of the **ioctl** subroutine.

Description

The **setioctlrv** subroutine sets the value of the u_ioctlrv field in the **uthread** structure of the running thread. The value in the u_ioctlrv field is returned by the**ioctl** or **fp_ioctl** subroutine on a successful completion. If the **ioctl** subroutine fails, an errno value is returned instead.

Return Values

The **setioctlrv** subroutine returns no return values.

Error Codes

The **setioctlrv** subroutine returns no error codes.

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

Item	Description
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

Item	Description
Nonzero value	Indicates that a **longjmpx** call caused the **setjmpx** service to return.
0	Indicates any other circumstances.

Related reference:

"clrjmpx Kernel Service" on page 43

Related information:

Handling Signals While in a System Call

Exception Processing

setpinit Kernel Service

Purpose

Sets the parent of the current kernel process to the initialization process.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/device.h>
int setpinit()
```

Description

The **setpinit** kernel service can be called by a kernel process to set its parent process to the **init** process. This is done to redirect the death of child signal for the termination of the kernel process. As a result, the init process is allowed to perform its default zombie process cleanup.

The **setpinit** service is used by a kernel process that can terminate, but does not want the user-mode process under which it was created to receive a death of child process notification.

Execution Environment

The **setpinit** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates that the current process is not a kernel process.

Related information:

Using Kernel Processes

Process and Exception Management Kernel Services

setuerror Kernel Service

Purpose

Allows kernel extensions to set the **ut_error** field for the current thread.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
```

```
int setuerror ( errno)
int errno;
```

Parameter

Item	Description
errno	Contains a value found in the **/usr/include/sys/errno.h** file that is to be copied to the current thread **ut_error** field.

Description

The **setuerror** kernel service allows a kernel extension in a process environment to set the **ut_error** field in current thread's **uthread** structure. Kernel extensions providing system calls available to user-mode applications typically use this service. For system calls, the value of the **ut_error** field in the per thread **uthread** structure is copied to the **errno** global variable by the system call handler before returning to the caller.

Execution Environment

The **setuerror** kernel service can be called from the process environment only.

Return Codes

The **setuerror** kernel service returns the *errno* parameter.

Related reference:

"getuerror Kernel Service" on page 190

Related information:

Kernel Extension and Device Driver Management Kernel Services

Understanding System Call Execution

shutdown_notify_reg Kernel Service

Purpose

Allows kernel extensions to register a shutdown notification.

Syntax

```
#include <sys/reboot.h>

int shutdown_notify_reg(sn)
shutdown_notify_t *sn;

typedef struct _shutdown_notify {
 struct _shutdown_notify *next;  /* Next in the link-list */
 int   version;  /* Version of structure */
 int   oper;     /* Bit map of the operation being performed */
 int   status;  /* The current status of this notify */
 uchar   cb_retry; /* Internal use */
 uchar   scope;  /* Partition or system wide */
 uchar   reason;  /* User initiated or EPOW */
 uchar   padding;    /* padding */
 long    (*func)(); /* Function kernel calls to notify ext. */
 void    *uaddr;
/* Address to help extension identify the object this structure refers to */
} shutdown_notify_t;

/* Valid values
for shutdown_notify_t->oper */
#define SHUTDOWN_NOTIFY_PREPARE 0x1 /* Shutdown has started */
#define SHUTDOWN_NOTIFY_REBOOT 0x2 /*
```

```
Final notify that shutdown will be a reboot */
#define SHUTDOWN_NOTIFY_HALT 0x4
/* Final notify that shutdown will be a halt */
#define SHUTDOWN_NOTIFY_QUERY 0x8
/* Check to see if finished shutdown */

/* Valid values for
shutdown_notify_t->status and
for SHUTDOWN_NOTIFY_QUERY return code */
#define SHUTDOWN_STATUS_PREPARE  0x1 /* Preparing for shutdown */
#define SHUTDOWN_STATUS_COMMENCE  0x2 /* Commencing shutdown */
#define SHUTDOWN_STATUS_FINISH  0x4 /* Finished shutdown */

#define SHUTDOWN_NOTIFY_VERSION  1   /* Increment by 1
           * every time add more
           * variables to
           * shutdown_notify_t
           */
/* Valid values for shutdown_notify_t->scope */
#define  SHUTDOWN_SCOPE_PARTITION 1
#define  SHUTDOWN_SCOPE_SYSTEM  2

/* Valid values for shutdown_notify_t->reason */
#define  SHUTDOWN_REASON_USER  1
#define  SHUTDOWN_REASON_EPOW  2

/* Valid handler return codes
during the SHUTDOWN_NOTIFY_PREPARE phase */
#define  SHUTDOWN_RC_SUCCESS     0
#define  SHUTDOWN_RC_DELAY   1

#define   SHUTDOWN_NOTIFY_VERSION  2
```

Description

The shutdown notify subsystem has been extended to provide additional information during a shutdown operation. During the **SHUTDOWN_NOTIFY_PREPARE** phase, the kernel provides information on the scope and reason for the shutdown action. Additionally, when a handler is called, before its completion, it can now delay the shutdown operation in order to finalize any outstanding jobs. The kernel again then calls out to the handler after some small amount of time. This process continues until all handlers return **SHUTDOWN_RC_SUCCESS**. This functionality is only present for **shutdown_notify_t** version 2 and preceding handlers. For version 1 handlers, the new fields are not present and the return code from the handler is ignored.

Parameters

Item	**Description**
cb_retry	Internal use.
func	Pointer to the function called to notify registered extension.
next	Pointer to next **shutdown_notify_t** structure in list.
oper	Bit map of operation(s) being performed.
padding	Padding.
reason	User initiated or EPOW event.
scope	Shutdown at the partition or system level.
sn	Pointer to a structure that the calling extension fills out when it registers.
status	Current status of notify.
uaddr	Place for extension to store an address to help it identify the object to which this structure refers.
version	Version of structure. Set to 1.
SHUTDOWN_NOTIFY_HALT	A halt is occurring.
SHUTDOWN_NOTIFY_PREPARE	Shutdown has started.
SHUTDOWN_NOTIFY_QUERY	Check to see if finished shutdown.

Item	Description
SHUTDOWN_NOTIFY_REBOOT	A reboot is occurring.
SHUTDOWN_NOTIFY_VERSION	Version number of structure.
SHUTDOWN_RC_DELAY	Return from registered handler to indicate its processing is not complete and wants to delay the shutdown operation.
SHUTDOWN_RC_SUCCESS	Return from registered handler to indicate all processing is complete and the shutdown operation can proceed.
SHUTDOWN_REASON_EPOW	EPOW event.
SHUTDOWN_REASON_USER	User initiated shutdown.
SHUTDOWN_SCOPE_PARTITION	Shutdown at the partition level.
SHUTDOWN_SCOPE_SYSTEM	Shutdown at the system level.
SHUTDOWN_STATUS_COMMENCE	Wrap up shutdown.
SHUTDOWN_STATUS_FINISH	Shutdown has completed.
SHUTDOWN_STATUS_PREPARE	Preparing for shutdown.

Execution Environment

Process environment only.

Return Values

Item	Description
0	Success.
EPERM	Attempted to register after prepare notification has started.
EINVAL	Invalid argument passed.

Related reference:
"shutdown_notify_unreg Kernel Service"

shutdown_notify_unreg Kernel Service

Purpose

Unregisters an extension from getting notified in the event of a shutdown.

Syntax

```
#include <sys/reboot.h>

int shutdown_notify_unreg(sn)
shutdown_notify_t *sn;
```

Description

The **shutdown_notify_unreg** kernel service unregisters an extension from getting notified in the event of a shutdown. The extension passes in the **shutdown_notify_t** instance it wants to unregister. This function will fail if it is called after the **SHUTDOWN_NOTIFY_HALT** and **SHUTDOWN_NOTIFY_REBOOT** notification process has started.

Parameters

Item	Description
sn	Pointer to a structure that the calling extension wants to unregister.

Execution Environment

Process environment only.

Return Values

Item	Description
0	Success
EPERM	Attempted to unregister after final notification has started.
EINVAL	Invalid argument passed.

Related reference:

"shutdown_notify_reg Kernel Service" on page 469

sig_chk Kernel Service

Purpose

Provides a kernel process the ability to poll for receipt of signals.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/signal.h>
int sig_chk ()
```

Description

> **Attention:** A system crash will occur if the **sig_chk** service is not called by a kernel process.

The **sig_chk** kernel service can be called by a kernel thread in kernel mode to determine if any unmasked signals have been received. Signals do not preempt threads because serialization of critical data areas would be lost. Instead, threads must poll for signals, either periodically or after a long sleep has been interrupted by a signal.

The **sig_chk** service checks for any pending signal that has a specified *signal catch* or *default* action. If one is found, the service returns the signal number as its return value. It also removes the signal from the pending signal mask. If no signal is found, this service returns a value of 0. The **sig_chk** service does not return signals that are blocked or ignored. It is the responsibility of the kernel process to handle the signal appropriately.

For kernel-only threads, the **sig_chk** kernel service clears the returned signal from the list of pending signals. For other kernel threads, the signal is not cleared, but left pending. It will be delivered to the kernel thread as soon as it returns to the user mode.

Understanding Kernel Threads in *Kernel Extensions and Device Support Programming Concepts* provides more information about kernel-only thread signal handling.

Execution Environment

The **sig_chk** kernel service can be called from the process environment only.

Return Values

Upon completion, the **sig_chk** service returns a value of 0 if no pending unmasked signal is found. Otherwise, it returns a nonzero signal value indicating the number of the highest priority signal that is pending. Signal values are defined in the **/usr/include/sys/signal.h** file.

Related information:

Introduction to Kernel Processes

Process and Exception Management Kernel Services

simple_lock or simple_lock_try Kernel Service

Purpose

Locks a simple lock.

Syntax

```
#include <sys/lock_def.h>

void simple_lock ( lock_addr)
simple_lock_t lock_addr;

boolean_t simple_lock_try ( lock_addr)
simple_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word to lock.

Description

The **simple_lock** kernel service locks the specified lock; it blocks if the lock is busy. The lock must have been previously initialized with the **simple_lock_init** kernel service. The **simple_lock** kernel service has no return values.

The **simple_lock_try** kernel service tries to lock the specified lock; it returns immediately without blocking if the lock is busy. If the lock is free, the **simple_lock_try** kernel service locks it. The lock must have been previously initialized with the **simple_lock_init** kernel service.

Note: When using simple locks to protect thread-interrupt critical sections, it is recommended that you use the **disable_lock** kernel service instead of calling the **simple_lock** kernel service directly.

Execution Environment

The **simple_lock** and **simple_lock_try** kernel services can be called from the process environment only.

Return Values

The **simple_lock_try** kernel service has the following return values:

Item	Description
TRUE	Indicates that the simple lock has been successfully acquired.
FALSE	Indicates that the simple lock is busy, and has not been acquired.

Related reference:

"disable_lock Kernel Service" on page 76

"simple_unlock Kernel Service" on page 475

Related information:

Understanding Locking

Locking Kernel Services

simple_lock_init Kernel Service

Purpose

Initializes a simple lock.

Syntax

```
#include <sys/lock_def.h>

void simple_lock_init ( lock_addr)
simple_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word.

Description

The **simple_lock_init** kernel service initializes a simple lock. This kernel service must be called before the simple lock is used. The simple lock must previously have been allocated with the **lock_alloc** kernel service.

Execution Environment

The **simple_lock_init** kernel service can be called from the process environment only.

The **simple_lock_init** kernel service may be called either the process or interrupt environments.

Return Values

The **simple_lock_init** kernel service has no return values.

Related reference:

"lock_alloc Kernel Service" on page 337

"simple_lock or simple_lock_try Kernel Service" on page 473

"simple_unlock Kernel Service" on page 475

Related information:

Understanding Locking

Locking Kernel Services

simple_unlock Kernel Service

Purpose

Unlocks a simple lock.

Syntax

```
#include <sys/lock_def.h>

void simple_unlock ( lock_addr)
simple_lock_t lock_addr;
```

Parameter

Item	Description
lock_addr	Specifies the address of the lock word to unlock.

Description

The **simple_unlock** kernel service unlocks the specified simple lock. The lock must be held by the thread which calls the **simple_unlock** kernel service. Once the simple lock is unlocked, the highest priority thread (if any) which is waiting for it is made runnable, and may compete for the lock again. If at least one kernel thread was waiting for the lock, the priority of the calling kernel thread is recomputed.

Note: When using simple locks to protect thread-interrupt critical sections, it is recommended that you use the **unlock_enable** kernel service instead of calling the **simple_unlock** kernel service directly.

Execution Environment

The **simple_unlock** kernel service can be called from the process environment only.

Return Values

The **simple_unlock** kernel service has no return values.

Related reference:

"simple_lock_init Kernel Service" on page 474

"simple_lock or simple_lock_try Kernel Service" on page 473

"unlock_enable Kernel Service" on page 517

Related information:

Understanding Locking

sleep Kernel Service

Purpose

Forces the calling kernel thread to wait on a specified channel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/pri.h>
#include <sys/proc.h>

int sleep ( chan,  priflags)
void *chan;
int priflags;
```

Parameters

Item	Description
chan	Specifies the channel number. For the **sleep** service, this parameter identifies the channel to wait for (sleep on).
priflags	Specifies two conditions: • The priority at which the kernel thread is to run when it is reactivated. • Flags indicating how a signal is to be handled by the **sleep** kernel service.

The valid flags and priority values are defined in the **/usr/include/sys/pri.h** file.

Description

The **sleep** kernel service is provided for compatibility only and should not be invoked by new code. The **e_sleep_thread** or **et_wait** kernel service should be used when writing new code.

The **sleep** service puts the calling kernel thread to sleep, causing it to wait for a wakeup to be issued for the channel specified by the *chan* parameter. When the process is woken up again, it runs with the priority specified in the *priflags* parameter. The new priority is effective until the process returns to user mode.

All processes that are waiting on the channel are restarted at once, causing a race condition to occur between the activated threads. Thus, after returning from the **sleep** service, each thread should check whether it needs to sleep again.

The channel specified by the *chan* parameter is simply an address that by convention identifies some event to wait for. When the kernel or kernel extension detects such an event, the **wakeup** service is called with the corresponding value in the *chan* parameter to start up all the threads waiting on that channel. The channel identifier must be unique systemwide. The address of an external kernel variable (which can be defined in a device driver) is generally used for this value.

If the **SWAKEONSIG** flag is not set in the *priflags* parameter, signals do not terminate the sleep. If the **SWAKEONSIG** flag is set and the **PCATCH** flag is not set, the kernel calls the **longjmpx** kernel service to resume the context saved by the last **setjmpx** call if a signal interrupts the sleep. Therefore, any system call (such as those calling device driver **ddopen**, **ddread**, and **ddwrite** routines) or kernel process that does an interruptible sleep without the **PCATCH** flag set must have set up a context using the **setjmpx** kernel service. This allows the sleep to resume in case a signal is sent to the sleeping process.

> **Attention:** The caller of the **sleep** service must own the kernel-mode lock specified by the *kernel_lock* parameter. The **sleep** service does not provide a compatible level of serialization if the kernel lock is not owned by the caller of the **sleep** service.

Execution Environment

The **sleep** kernel service can be called from the process environment only.

Return Values

Item	Description
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.

Related information:

Locking Strategy in Kernel Mode

Process and Exception Management Kernel Services

sleepx Kernel Service

Purpose

Wait for an event.

Syntax

#include `<sys/sleep.h>`

int sleepx (*tchan_t chan int pri flags_t flags***)**

Parameters

`chan`
: Specifies the channel number. For the **sleep** service, this parameter identifies the channel to wait for (sleep on).

`pri`
: Specifies the wakeup priority

`flags`
: Signal control flags

Description

Wait for an event to occur. This procedure can only be called by a thread. Callers of this service must be prepared for a premature return and check that the reason for waiting has gone away.

The **`pri`** parameter will be the priority of the thread when it becomes runnable again (if that priority is more favorable). The process will keep that priority until it is dispatched. The range of the wakeup priority is `0 <= pri <= PRI_LOW`. If the **`pri`** parameter is outside of that range, it is forced to the lower or upper boundary.

Execution Environment

The **sleepx** kernel service can be called from the process environment only.

Return Values

0	Indicates that the event occurred.
1	Indicates that the event signalled out.

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

Item	Description
uaddr	Specifies the address of user data.
c	Specifies the character to store.

Description

The **subyte** kernel service stores a byte of data at the specified address in user memory. It is provided so that system calls and device heads can safely access user data. The **subyte** service ensures that the user has the appropriate authority to:

- Access the data.
- Protect the operating system from paging I/O errors on user data.

The **subyte** service should only be called while executing in kernel mode in the user process.

Execution Environment

The **subyte** kernel service can be called from the process environment only.

Return Values

Item	Description
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.

Related reference:

"fubyte Kernel Service" on page 178

Related information:

Accessing User-Mode Data While in Kernel Mode

Memory Kernel Services

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

Item	Description
ep	Points to a character variable where the **EPERM** value is stored on failure.

Description

The **suser** kernel service checks whether a process has any effective privilege (that is, whether the process's `uid` field equals 0).

Execution Environment

The **suser** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates failure. The character pointed to by the *ep* parameter is set to the value of **EPERM**. This indicates that the calling process does not have any effective privilege.
Nonzero value	Indicates success (the process has the specified privilege).

Related information:

Security Kernel Services

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

Item	Description
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

Item	Description
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.

Related reference:

"fuword Kernel Service" on page 179

Related information:

Memory Kernel Services

Accessing User-Mode Data While in Kernel Mode

t

The following kernel services begin with the with the letter t.

TE_verify_reg Kernel Service

Purpose

Registers a callout handler for Trusted Execution (TE) file verification during the exec() functions, kernel extension loads, and library load operations.

Syntax

```
#include <sys/file.h>
typedef int (*TE_verify)(char *, int, struct file *);

int TE_verify_reg(TE_verify verify_fn, uint_64 options)
```

Parameters

`verify_fn`

Specifies the callout function to be called for the verification checks with the exec() functions for the Trusted Execution of the AIX kernel level, loading of kernel extensions, and library loading events instead of the default AIX Trusted Execution method.

For more information about the function definition of this callout handler, see the `alt_verify_fn` section.

`options`

Specifies a bit mask of registration options. The **`options`** parameter is not defined currently. The caller must set the **`options`** parameter to `0`.

Description

The `TE_verify_reg` kernel service registers a callout handler for the AIX Trusted Execution framework.

After a callout handler is registered, the handler is invoked for the exec() functions, loading kernel extensions, and library load-time checks for Trusted Execution in the AIX kernel. The default AIX Trusted Execution logic is not invoked and any AIX-configured policies for Trusted Execution not applied. The registered alternative handler becomes the active Trusted Execution engine for AIX to provide security policy as implemented in the handler and its associated management components.

After a callout handler is registered with the TE_verify_reg kernel service, subsequent invocation of the TE_verify_reg service returns with an error code of EEXIST.

You must have root authority to call the TE_verify_reg kernel service.

Return values

On successful completion, the TE_verify_reg service kernel service returns a value of 0.

The following error codes are returned on failure:

EEXIST
The callout handler is already registered.

EPERM
The caller does not have permission to invoke this function.

EINVAL
The callout handler or the **options** parameters are invalid.

Execution environment

The TE_verify_reg kernel service can be called from the process environment only.

The registered alternative Trusted Execution handler must conform to the behaviors that are described in the following section.

alt_verify_fn callout function

Purpose

Verifies the integrity of a file.

Syntax

```
#include <sys/file.h>

#define VERIFY_EXECUTABLES 2
#define VERIFY_SHLIBS 3
#define VERIFY_SCRIPTS 4
#define VERIFY_KERNEXTS 5

int alt_TE_verify (char *path_name, int type, struct file *path_fp)
```

Description

The alt_TE_verify callout function is started from the loader and the program execution path to verify the integrity of a file that is specified under the *path_name* parameter. The *path_fp* parameter is a file pointer to the file object that is associated with the *path_name* parameter.

The *type* parameter can be one of the following values:

VERIFY_EXECUTABLES
This value is specified when the alt_TE_verify function is started from the kernel exec() function to verify executable programs.

VERIFY_SCRIPTS
This value is specified when the alt_TE_verify function is started from the exec() function and the *path_name* value is a shell file.

VERIFY_KERNEXTS
This value is specified when the `alt_TE_verify` function is started for loading a kernel extension.

VERIFY_SHLIBS
This value is specified when the `alt_TE_verify` function is started for loading a shared library.

Input parameters

`path_name`
Specifies the path to the file that must be verified.

`type`
Indicates the type of verification that must be performed.

`path_fp`
Indicates the file pointer to the *path_name* file.

Return values

`0` Indicates that the verification completed successfully.

`Nonzero`
Indicates that the verification failed.

The nonzero return value blocks loading of the file. An error number is set by the AIX kernel functions that start the `alt_verify_fn` callout function.

TE_verify_unreg Kernel Service

Purpose

Unregisters a previously registered callout handler for trusted execution.

Syntax

```
#include <sys/file.h>
typedef int (*TE_verify)(char *, int, struct file *);

int TE_verify_unreg(TE_verify verify_fn, uint_64 options)
```

Parameters

`verify_fn`
Specifies the callout function that must be used when you register the handler by using the `TE_verify_reg()` kernel service.

`options`
Specifies a bit mask of registration options. The **`options`** parameter is not defined currently. The caller must set the **`options`** parameter to `0`.

Description

The `TE_verify_unreg` kernel service unregisters a callout handler for the AIX Trusted Execution (TE) framework. The *verify_fn* parameter must match with the currently registered TE callout handler. Otherwise, the `TE_verify_unreg` kernel service returns an error code of `EPERM`.

After a callout handler is unregistered, the default AIX trusted execution logic is applied based on the configured AIX trusted execution policies.

The caller of the `TE_verify_unreg` kernel service must have root authority.

Return values

On successful completion, the `TE_verify_unreg` kernel service returns a value of 0.

The following error codes are returned on failure:

EPERM
The caller does not have permission to start this function. Or, the registered callout handler is not same as the *verify_fn* parameter.

EINVAL
No callout handler is registered or the *options* parameters are invalid.

Execution environment

The `TE_verify_unreg` kernel service can be called only from the process environment.

talloc Kernel Service

Purpose

Allocates a timer request block before starting a timer request.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/timer.h>
```

```
struct trb *talloc()
```

Description

The **talloc** kernel service allocates a timer request block. The user must call it before starting a timer request with the **tstart** kernel service. If successful, the **talloc** service returns a pointer to a pinned timer request block.

Execution Environment

The **talloc** kernel service can be called from the process environment only.

Return Values

The **talloc** service returns a pointer to a timer request block upon successful allocation of a **trb** structure. Upon failure, a null value is returned.

Related reference:

"tfree Kernel Service"

Related information:

Timer and Time-of-Day Kernel Services

Using Fine Granularity Timer Services and Structures

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

Item	Description
t	Points to the timer request structure to be freed.

Description

The **tfree** kernel service deallocates a timer request block that was previously allocated with a call to the **talloc** kernel service. The caller of the **tfree** service must first cancel any pending timer request associated with the timer request block being freed before attempting to free the request block. Canceling the timer request block can be done using the **tstop** kernel service.

Execution Environment

The **tfree** kernel service can be called from either the process or interrupt environment.

Note: Do not use the **tfree** kernel service to free the timer request block that is passed to the timer completion handler.

Return Values

The **tfree** service has no return values.

Related reference:

"talloc Kernel Service" on page 483

Related information:

Timer and Time-of-Day Kernel Services

Using Fine Granularity Timer Services and Structures

thread_create Kernel Service

Purpose

Creates a new kernel thread in the calling process.

Syntax

```
#include <sys/thread.h>

tid_t thread_create ()
```

Description

The **thread_create** kernel service creates a new kernel-only thread in the calling kernel process. The thread's ID is returned; it is unique system wide.

The new thread does not begin running immediately; its state is set to **TSIDL**. The execution will start after a call to the **kthread_start** kernel service. If the process is exited prior to the thread being made runnable, the thread's resources are released immediately. The thread's signal mask is inherited from the calling thread; the set of pending signals is cleared. Signals sent to the thread are marked pending while the thread is in the **TSIDL** state.

If the calling thread is bound to a specific processor, the new thread will also be bound to the processor.

Execution Environment

The **thread_create** kernel service can be called from the process environment only. This service cannot be called directly from a kernel extension.

Return Values

Upon successful completion, the new thread's ID is returned. Otherwise, -1 is returned, and the error code can be checked by calling the **getuerror** kernel service.

Error Codes

Item	Description
EAGAIN	The total number of kernel threads executing system wide or the maximum number of kernel threads per process would be exceeded.
ENOMEM	There is not sufficient memory to create the kernel thread.
ENOTSUP	The **thread_create** service was called directly from a kernel extension.

Related reference:

“kthread_start Kernel Service” on page 310

Related information:

Process and Exception Management Kernel Services

thread_self Kernel Service

Purpose

Returns the caller's kernel thread ID.

Syntax

```
#include <sys/thread.h>
tid_t thread_self ()
```

Description

The **thread_self** kernel service returns the thread process ID of the calling process.

The **thread_self** service can also be used to check the environment that the routine is being executed in. If the caller is executing in the interrupt environment, the **thread_self** service returns a process ID of -1. If a routine is executing in a process environment, the **thread_self** service obtains the thread process ID.

Execution Environment

The **thread_self** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
-1	Indicates that the **thread_self** service was called from an interrupt environment.

The **thread_self** service returns the thread process ID of the current process if called from a process environment.

Related information:

Process and Exception Management Kernel Services

Understanding Execution Environments

thread_setsched Kernel Service

Purpose

Sets kernel thread scheduling parameters.

Syntax

```
#include <sys/thread.h>
#include <sys/sched.h>

int thread_setsched ( tid,  priority,  policy)
tid_t tid;
int priority;
int policy;
```

Parameters

Item	Description
tid	Specifies the kernel thread.
priority	Specifies the priority. It must be in the range from 0 to **PRI_LOW**; 0 is the most favored priority.
policy	Specifies the scheduling policy. It must have one of the following values: **SCHED_FIFO** Denotes fixed priority first-in first-out scheduling. **SCHED_FIFO2** Allows a thread that sleeps for a relatively short amount of time to be requeued to the head, rather than the tail, of its priority run queue. **SCHED_FIFO3** Causes threads to be enqueued to the head of their run queues. **SCHED_RR** Denotes fixed priority round-robin scheduling. **SCHED_OTHER** Denotes the default scheduling policy.

Description

The **thread_setsched** subroutine sets the scheduling parameters for a kernel thread. This includes both the priority and the scheduling policy, which are specified in the *priority* and *policy* parameters. The calling and the target thread must be in the same process.

When setting the scheduling policy to **SCHED_OTHER**, the system chooses the priority; the *priority* parameter is ignored. The only way to influence the priority of a thread using the default scheduling policy is to change the process nice value.

The calling thread must belong to a process with root authority to change the scheduling policy of a thread to either **SCHED_FIFO**, **SCHED_FIFO2**, **SCHED_FIFO3**, or **SCHED_RR**.

Execution Environment

The **thread_setsched** kernel service can be called from the process environment only.

Return Values

Upon successful completion, 0 is returned. Otherwise, -1 is returned, and the error code can be checked by calling the **getuerror** kernel service.

Error Codes

Item	Description
EINVAL	The *priority* or *policy* parameters are not valid.
EPERM	The calling kernel thread does not have sufficient privilege to perform the operation.
ESRCH	The kernel thread *tid* does not exist.

Related reference:

"thread_create Kernel Service" on page 484

Related information:

Process and Exception Management Kernel Services

thread_set_smt_priority or thread_read_smt_priority System Call

Purpose

Sets or reads the current simultaneous multithreading (SMT) thread priority for a user-thread.

Syntax

```
#include <sys/errno.h>
#include <sys/thread.h>
#include <sys/processor.h>

int thread_set_smt_priority ( Priority )
smt_thread_priority_t Priority;
```

```
#include <sys/errno.h>
#include <sys/thread.h>
#include <sys/processor.h>

smt_thread_priority_t thread_read_smt_priority ( )
```

Description

The SMT thread priority that is associated with a logical CPU, SMT hardware thread, controls the relative priority of the logical CPU in relation to the other logical CPUs on the same processor core. The relative priority between the SMT hardware threads on a processor core determines how decode cycles are granted to each SMT hardware thread. The SMT thread priority can be used to cause a particular application thread to be favored over other application threads that are running on the other SMT hardware threads in the same processor core. It is done by increasing the SMT thread priority of the logical CPU the application is running on, or by lowering the SMT thread priority of the application threads that are running on the other logical CPUs associated with the same processor core.

The **thread_set_smt_priority** and **thread_read_smt_priority** system calls provide a way to register and read back the current SMT thread priority on a per process-thread basis.

Note:

These interfaces are not supported on some processor architectures.

If the process-thread is dispatched to a logical CPU that is running in non-SMT mode, the SMT thread priority level has no effect.

Callers of the **thread_set_smt_priority** system call with normal user-level privileges can set their SMT thread priority level to one of the following levels:

- LOW
- MEDIUM LOW
- NORMAL

Callers that have RBAC PV_PROC_VARS privilege can set their priority level to one of the following levels:

- VERY LOW
- LOW
- MEDIUM LOW
- NORMAL
- MEDIUM HIGH
- HIGH

The default thread priority level is NORMAL.

Note: The only supported means for altering the SMT thread priority level is by using the **thread_set_smt_priority** system call. If an alternative means of setting the SMT priority is used, the kernel does not know the process-thread's current SMT priority level, and overwrites the required SMT priority level without restoring it.

The **thread_read_smt_priority** system call returns the current SMT priority level that is registered by the process thread. If the process thread did not register a required SMT priority level, then the default priority level of NORMAL is returned.

Parameters

Item	Description
Priority	Used to specify one of the following parameters: • `T_VERYLOW_SMT_PRI` • `T_LOW_SMT_PRI` • `T_MEDIUMLOW_SMT_PRI` • `T_NORMAL_SMT_PRI` • `T_MEDIUMHIGH_SMT_PRI` • `T_HIGH_SMT_PRI`

Execution Environment

The **thread_read_smt_priority** and **thread_set_smt_priority** system calls can be called from the process environment only.

Return Values

On successful completion, the **thread_set_smt_priority** system call returns 0. Otherwise, **-1** is returned and the **errno** global variable is set to indicate the error.

On successful completion, the **thread_read_smt_priority** system call returns the current required SMT priority. Otherwise, **-1** is returned and the **errno** global variable is set to indicate the error.

Error Codes

Item	Description
EPERM	The process attempted to set the SMT thread priority level to a value other than T_LOW_SMT_PRI, T_MEDIUMLOW_SMT_PRI, or T_NORMAL_SMT_PRI and does not have the necessary privileges.
EINVAL	The required priority value that is specified is invalid.
ENOSYS	SMT thread priority level manipulation is not supported on this system.

thread_terminate Kernel Service

Purpose

Terminates the calling kernel thread.

Syntax

```
#include <sys/thread.h>
void thread_terminate ()
```

Description

The **thread_terminate** kernel service terminates the calling kernel thread and cleans up its structure and its kernel stack. If it is the last thread in the process, the process will exit.

The **thread_terminate** kernel service is automatically called when a thread returns from its entry point routine (defined in the call to the **kthread_start** kernel service).

Execution Environment

The **thread_terminate** kernel service can be called from the process environment only.

Return Values

The **thread_terminate** kernel service never returns.

Related reference:

"kthread_start Kernel Service" on page 310

Related information:

Process and Exception Management Kernel Services

timeout Kernel Service

Attention: This service must not be used because it is not multi-processor safe. The base kernel timer and watchdog services must be used instead.

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

Item	Description
func	Indicates the function to be called.
arg	Indicates the parameter to supply to the function specified by the *func* parameter.
ticks	Specifies the number of timer ticks that must occur before the function specified by the *func* parameter is called. Many timer ticks can occur per second. The HZ label that is found in the **/usr/include/sys/m_param.h** file can be used to determine the number of ticks per second.

Description

The **timeout** service is not part of the kernel. However, it is a compatibility service that is provided in the **libsys.a** library. To use the **timeout** service, a kernel extension must be bound with the **libsys.a** library. The **timeout** service, like the associated kernel services **untimeout** and **timeoutcf**, can be bound and used only in the pinned part of a kernel extension or the bottom half of a device driver because these services use interrupt disable for serialization.

The **timeout** service schedules the function pointed to by the *func* parameter to be called with the *arg* parameter after the number of timer ticks that are 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 the **timeout** service is called.

Calling the **timeout** service without allocating enough callout table entries can result in a kernel panic because of a lack of pinned callout table elements. The value of a timer tick depends on the hardware's capability. You can use the **restimer** subroutine to determine the minimum granularity.

Multiple pending **timeout** requests with the same *func* and *arg* parameters are not allowed.

The func Parameter

The function that is specified by the *func* parameter must be declared as follows:

```
void func (arg)
void *arg;
```

Execution Environment

The **timeout** routine can be called from either the process or interrupt environment.

The function that is specified by the *func* parameter is called in the interrupt environment. Therefore, it must follow the conventions for interrupt handlers.

Return Values

The **timeout** service has no return values.

Related reference:

"untimeout Kernel Service" on page 522

"timeoutcf Subroutine for Kernel Services" on page 491

Related information:

restimer subroutine

Timer and Time-of-Day Kernel Services

timeoutcf Subroutine for Kernel Services

Attention: This service must not be used because it is not multi-processor safe. The base kernel timer and watchdog services must be used instead.

Purpose

Allocates or deallocates callout table entries for use with the **timeout** kernel service.

Library

libsys.a (Kernel extension runtime routines)

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int timeoutcf ( cocnt)
int cocnt;
```

Parameter

Item	Description
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 that is provided in the **libsys.a** library. To use the **timeoutcf** subroutine, a kernel extension must be bound with the **libsys.a** library. The **timeoutcf** subroutine, like the associated kernel **libsys** services **untimeout** and **timeout**, can be bound and used only in the pinned part of a kernel extension or the bottom half of a device driver because these services use interrupt disable for serialization.

The **timeoutcf** subroutine registers an increase or decrease in the number of callout table entries available for the **timeout** subroutine to use. Before a subroutine can use the **timeout** kernel service, the **timeoutcf** subroutine must increase the number of callout table entries available to the **timeout** kernel service. It increases this number by the maximum number of outstanding time outs that the routine can have pending at one time.

The **timeoutcf** subroutine must be used to decrease the number of callout table entries by the amount it was increased under the following conditions:

- The routine that uses the **timeout** subroutine finished using it.
- The calling routine has no more outstanding timeout requests that are pending.

Typically the **timeoutcf** subroutine is called in a device driver's **open** and **close** routine. It is called to allocate and deallocate sufficient elements for the maximum expected use of the **timeout** kernel service for that instance of the open device.

Attention: A kernel panic results either of these two circumstances:

- A request to decrease the callout table allocation is made that is greater than the number of unused callout table entries.
- The **timeoutcf** subroutine is called in an interrupt environment.

Execution Environment

The **timeoutcf** subroutine can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful allocation or deallocation of the requested callout table entries.
-1	Indicates an unsuccessful operation.

Related reference:

"timeout Kernel Service" on page 489

Related information:

Timer and Time-of-Day Kernel Services

trc_ishookon Exported Kernel Service

Purpose

Checks if a given trace hook word is being traced by system trace.

Syntax

```
#include <sys/trcmacros.h>

int trc_ishookon (int chan, long hkwd);
```

Description

The **trc_ishookon** kernel service informs the user if tracing is on and the specified hook word is being traced.

Parameters

Item	Description
chan	The channel to query with the range from 0 to 7.
hkwd	The hook word to be traced by system trace.

Return Values

Item	Description
1	The hook word is being traced.
0	Hook word is not being traced or system trace is off.

Related information:

trace subroutine

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

Item	Description
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** Before AIX 6.1 the hook identifier is a 12-bit value. On AIX 6.1 and above, the hook identifier is a 16-bit value. A 16-bit value of the form hhh0 is equivalent to a 12-bit value of the form hhh. **hk_type** A 4-bit hook type. The **trcgenk** service automatically records this information. This value is only valid before AIX 6.1.
data_word	Specifies a word of user-defined data.
len	Specifies the length in bytes of the buffer specified by the *buf* parameter.
buf	Points to a buffer of trace data. The maximum amount of trace data is 4096 bytes.

Description

The **trcgenk** kernel service records a trace event if a trace session is active for the specified trace channel. If a trace session is not active, the **trcgenk** kernel service simply returns. The **trcgenk** kernel service is located in pinned kernel memory.

The **trcgenk** kernel service is used to record a trace entry consisting of an *hk_word* entry, a *data_word* entry, a variable number of bytes of trace data, and, in AIX 5L™ Version 5.3 with the 5300-05 Technology Level and above, a time stamp.

Execution Environment

The **trcgenk** kernel service can be called from either the process or interrupt environment.

Return Values

The **trcgenk** kernel service has no return values.

Related reference:

"trcgenkt Kernel Service"

Related information:

trace subroutine

trcgen subroutine

RAS Kernel Services

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

Item	Description
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** Before AIX 6.1 the hook identifier is a 12-bit value. On AIX 6.1 and above, the hook identifier is a 16-bit value. A 16-bit value of the form hhh0 is equivalent to a 12-bit value of the form hhh. **hk_type** A 4-bit hook type. The **trcgenkt** service automatically records this information. This value is only valid before AIX 6.1.
data_word	Specifies a word of user-defined data.
len	Specifies the length, in bytes, of the buffer identified by the *buf* parameter.
buf	Points to a buffer of trace data. The maximum amount of trace data is 4096 bytes.

Description

The **trcgenkt** kernel service records a trace event if a trace session is active for the specified trace channel. If a trace session is not active, the **trcgenkt** service simply returns. The **trcgenkt** kernel service is located in pinned kernel memory.

The **trcgenkt** service records a trace entry consisting of an *hk_word* entry, a *data_word* entry, a variable number of bytes of trace data, and a time stamp.

Execution Environment

The **trcgenkt** kernel service can be called from either the process or interrupt environment.

Return Values

The **trcgenkt** service has no return values.

Related reference:

"trcgenk Kernel Service" on page 492

Related information:

trace command

trcgen subroutine

RAS Kernel Services

trcgenkt Kernel Service for Data Link Control (DLC) Devices

Purpose

Records a trace event, including a time stamp, for a DLC trace channel.

Syntax

```
#include <sys/trchkid.h>

void trcgenkt (chan, hk_word, data_word, len, buf)
unsigned int  chan,  hk_word,  data_word,  len;
char * buf;
```

Parameters

Item **Description**

chan
Specifies the channel number for the trace session. This number is obtained from the **trcstart** subroutine.

hk_word
Contains the trace hook identifier defined in the **/usr/include/sys/trchkid.h** file. The types of link trace entries registered using the hook ID include:

HKWD_SYSX_DLC_START
Start link station completions

HKWD_SYSX_DLC_TIMER
Time-out completions

HKWD_SYSX_DLC_XMIT
Transmit completions

HKWD_SYSX_DLC_RECV
Receive completions

HKWD_SYSX_DLC_HALT
Halt link station completions

data_word
Specifies trace data format field. This field varies depending on the hook ID. Each of these definitions are in the **/usr/include/sys/gdlextcb.h** file:

- The first half-word always contains the data link protocol field including one of these definitions:

  DLC_DL_SDLC
  SDLC

  DLC_DL_HDLC
  HDLC

  DLC_DL_BSC
  BISYNC

  DLC_DL_ASC
  ASYNC

  DLC_DL_PCNET
  PC Network

  DLC_DL_ETHER
  Standard Ethernet

  DLC_DL_802_3
  IEEE 802.3

  DLC_DL_TOKEN
  Token-Ring

Item **Description**

- On start or halt link station completion, the second half-word contains the physical link protocol in use:

  DLC_PL_EIA232
  EIA-232D Telecommunications

  DLC_PL_EIA366
  EIA-366 Auto Dial

  DLC_PL_X21
  CCITT X.21 Data Network

  DLC_PL_PCNET
  PC Network Broadband

  DLC_PL_ETHER
  Standard Baseband Ethernet

  DLC_PL_SMART
  Smart Modem Auto Dial

  DLC_PL_802_3
  IEEE 802.3 Baseband Ethernet

  DLC_PL_TBUS
  IEEE 802.4 Token Bus

  DLC_PL_TRING
  IEEE 802.5 Token-Ring

  DLC_PL_EIA422
  EIA-422 Telecommunications

  DLC_PL_V35
  CCITT V.35 Telecommunications

  DLC_PL_V25BIS
  CCITT V.25 bis Autodial for Telecommunications

- On timeout completion, the second half-word contains the type of timeout occurrence:

  DLC_TO_SLOW_POLL
  Slow station poll

  DLC_TO_IDLE_POLL
  Idle station poll

  DLC_TO_ABORT
  Link station aborted

  DLC_TO_INACT
  Link station receive inactivity

  DLC_TO_FAILSAFE
  Command failsafe

  DLC_TO_REPOLL_T1
  Command repoll

  DLC_TO_ACK_T2
  I-frame acknowledgment

- On transmit completion, the second half-word is set to the data link control bytes being sent. Some transmit packets only have a single control byte; in that case, the second control byte is not displayed.
- On receive completion, the second half-word is set to the data link control bytes that were received. Some receive packets only have a single control byte; in that case, the second control byte is not displayed.

len Specifies the length in bytes of the entry specific data specified by the *buf* parameter.

Item	Description
buf	Specifies the pointer to the entry specific data that consists of: **Start Link Station Completions** Link station diagnostic tag and the remote station's name and address. **Time-out Completions** No specific data is recorded. **Transmit Completions** Either the first 80 bytes or all the transmitted data, depending on the short/long trace option. **Receive Completions** Either the first 80 bytes or all the received data, depending on the short/long trace option. **Halt Link Station Completions** Link station diagnostic tag, the remote station's name and address, and the result code.

Description

The **trcgenkt** kernel service records a trace event if a trace session is active for the specified trace channel. If a trace session is not active, the **trcgenkt** kernel service simply returns. The **trcgenkt** kernel service is located in pinned kernel memory.

The **trcgenkt** kernel service is used to record a trace entry consisting of an *hk_word* entry, a *data_word* entry, a variable number of bytes of trace data, and a time stamp.

Execution Environment

The **trcgenkt** kernel service can be called from either the process or interrupt environment.

Return Values

The **trcgenkt** kernel service has no return values.

Related reference:

“trcgenk Kernel Service” on page 492

“trcgenkt Kernel Service” on page 493

Related information:

trace subroutine

Generic Data Link Control (GDLC) Environment Overview

RAS Kernel Services

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

Item	Description
t	Points to a timer request structure.

Description

The **tstart** kernel service submits a timer request with the timer request block specified by the *t* parameter as input. The caller of the **tstart** kernel service must first call the **talloc** kernel service to allocate the timer request structure. The caller must then initialize the structure's fields before calling the **tstart** kernel service.

Once the request has been submitted, the kernel calls the `t->func` timer function when the amount of time specified by the `t->timeout.it` value has elapsed. The `t->func` timer function is called on an interrupt level. Therefore, code for this routine must follow conventions for interrupt handlers.

The **tstart** kernel service examines the `t->flags` field to determine if the timer request being submitted represents an absolute request or an incremental one. An absolute request is a request for a time out at the time represented in the **it_value** structure. An incremental request is a request for a time out at the time represented by now, plus the time in the **it_value** structure.

The caller should place time information for both absolute and incremental timers in the **itimerstruc_t t.it** value substructure. The **T_ABSOLUTE** absolute request flag is defined in the **/usr/include/sys/timer.h** file and should be ORed into the `t->flag` field if an absolute timer request is desired.

When the **T_MOVE_OK** flag is set, the associated timer is moved to another processor when the owning processor is folded.

When **T_LATE_OK** flag is set, the associated timer is put to sleep when the owning processor is put to sleep (folded) mode. The timer expiration handler is called when the owning processor is awakened (unfolded) if the scheduled expiration time has past. The time spent sleeping is therefore counted with respect to the expiration time. When this flag is set, there is no guarantee as to when the timer might expire.

Note: The **T_MOVE_OK** and **T_LATE_OK** flags are not required. They are intended to improve the effectiveness of processor folding by reducing the load on folded processors.

Modifications to the system time are added to incremental timer requests, but not to absolute ones. Consider the user who has submitted an absolute timer request for noon on 12/25/88. If a privileged user then modifies the system time by adding four hours to it, then the timer request submitted by the user still occurs at noon on 12/25/88.

By contrast, suppose it is presently 12 noon and a user submits an incremental timer request for 6 hours from now (to occur at 6 p.m.). If, before the timer expires, the privileged user modifies the system time by adding four hours to it, the user's timer request will then expire at 2200 (10 p.m.).

Execution Environment

The **tstart** kernel service can be called from either the process or interrupt environment.

Return Values

The **tstart** service has no return values.

Related reference:

"tstop Kernel Service" on page 499

Related information:

Timer and Time-of-Day Kernel Services

Using Fine Granularity Timer Services and Structures

tstop Kernel Service

Purpose

Cancels a pending timer request.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/timer.h>

int tstop ( t)
struct trb *t;
```

Parameter

Item	Description
t	Specifies the pending timer request to cancel.

Description

The **tstop** kernel service cancels a pending timer request. The **tstop** kernel service must be called before a timer request block can be freed with the **tfree** kernel service.

In a multiprocessor environment, the timer function associated with a timer request block may be active on another processor when the **tstop** kernel service is called. In this case, the timer request cannot be canceled. A multiprocessor-safe driver must therefore check the return code and take appropriate action if the cancel request failed.

In a uniprocessor environment, the call always succeeds. This is untrue in a multiprocessor environment, where the call will fail if the timer is being handled by another processor. Therefore, the function now has a return value, which is set to 0 if successful, or -1 otherwise. Funnelled device drivers do not need to check the return value since they run in a logical uniprocessor environment. Multiprocessor-safe and multiprocessor-efficient device drivers need to check the return value in a loop. In addition, if a driver uses locking, it must release and reacquire its lock within this loop. A delay should be used between the release and reacquiring the lock as shown below:

```
while (tstop(&trp)) {
       release_any_lock;
       delay_some_time;
       reacquire_the_lock;
} /* null while loop if locks not used */
```

Execution Environment

The **tstop** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
0	Indicates that the request was successfully canceled.
-1	Indicates that the request could not be canceled.

Related reference:

"tstart Kernel Service" on page 497

Related information:

Timer and Time-of-Day Kernel Services

Using Fine Granularity Timer Services and Structures

Using Multiprocessor-Safe Timer Services

tuning Kernel Service

Purpose

Provides access to the kernel tunable variables through an easily accessible interface.

Syntax

```
typedef enum {
    TH_MORE,
    TH_EOF
} tmode_t;

#define TH_ABORT TH_EOF

typedef int (*tuning_read_t)(tmode_t mode, long *size, char **buf, void *context);
typedef int (*tuning_write_t)(tmode_t mode, long *size, char *buf, void *context);

tinode_t *tuning_register_handler (path,  mode, readfunc, writefunc, context)
const char *path;
mode_t mode;
tuning_read_t readfunc;
tuning_write_t writefunc;
void * context;

tinode *tuning_register_bint32 (path, mode, variable, low,  high)
const char *path;
mode_t mode;
int32 *variable;
int32 low;
int32 high;

tinode *tuning_register_bint32x (path, rfunc, wfunc, mode, low, high)
const char *path;
mode_t mode;
int32 (*rfunc)(void *);
int (*wfunc)(int32, void *);
void *context;
int32 low;
int32 high;

tinode *tuning_register_buint32 (path, mode,variable, low, high)
const char *path;
mode_t mode;
uint32 *variable;
uint32 low;
uint32 high;

tinode *tuning_register_buint32x (path, rfunc, wfunc, mode, low, high)
const char *path;
mode_t mode;
uint32 (*rfunc)(void *);
int (*wfunc)(uint32, void *);
void *context;
uint32 low;
uint32 high;
```

```
tinode *tuning_register_bint64 (path, mode, variable, low, high)
const char *path;
mode_t mode;
int64 *variable;
int64 low;
int64 high;
```

```
tinode *tuning_register_bint64x (path, rfunc, wfunc, mode, low, high)
const char *path;
mode_t mode;
int64 (*rfunc)(void *);
int (*wfunc)(int64, void *);
void *context;
in64 low;
in64 high;
```

```
tinode *tuning_register_buint64 (path, mode, variable, low, high)
const char *path;
mode_t mode;
uint64 *variable;
uint64 low;
uint64 high;
```

```
tinode *tuning_register_buint64x (path, rfunc, wfunc, mode, low, high)
const char *path;
mode_t mode;
uint64 (*rfunc)(void *);
int (*wfunc)(uint64, void *);
void *context;
uint64 low;
uint64 high;
```

```
void tuning_deregister (t)
tinode_t * t;
```

Description

The **tuning_register_handler** kernel service is used to add a file at the location specified by the *path* parameter. When this file is read from or written to, one of the two callbacks passed as parameters to the function is invoked.

Accesses to the file are viewed in terms of streams. A single stream is created by a sequence of one open, one or more reads, and one close on the file. While the file is open by one process, attempts to open the same file by other processes will be blocked unless **O_NONBLOCK** is passed in the flags to the **open** subroutine.

The *readfunc* callback behaves like a producer function. The function is called when the user attempts to read from the file. The *mode* parameter is equal to **TH_MORE** unless the user closes the file prematurely. On entry, the *size* parameter is an integer containing the size of the buffer. The *context* parameter is the context pointer passed to the registration function. Upon return, *size* should contain either the actual amount of data returned, or a zero if an end-of-file condition should be returned to the user. The return value of the function can also be used to signal end-of-file, as described below.

Note: It is expected that the *readfunc* callback has already done any necessary end-of-file cleanup when it returns the end-of-file signal.

If the amount of data returned is nonzero, the *buf* parameter may be modified to point to a new buffer. If this is done, the callback is responsible for freeing the new buffer.

If the buffer provided by the caller is too small, the caller may instead set *buf* to NULL. In this case, the *size* parameter should be modified to indicate the size of the buffer needed. The caller will then re-invoke the callback with a buffer of at least the requested size.

If the user closes the file before the callback indicates end-of-file, the callback will be invoked one last time with *mode* equal to **TH_ABORT**. In this case, the *size* parameter is equal to 0 on entry, and any data returned is discarded. The callback must reset its state because no further callbacks will be made for this stream.

The *writefunc* callback behaves as a consumer function and is used when the user attempts to write to the file. The *mode* parameter is set to **TH_EOF** if no further data can be expected on this stream (for example, the user called the **close** subroutine on the file). Otherwise, *mode* is set to **TH_MORE**. The *size* parameter contains the size of the data passed in the buffer. The *buf* parameter is the pointer to the buffer.

Note: There will be zero or more calls with the *mode* parameter set to **TH_MORE** and one call with the *mode* parameter set to **TH_EOF** for every stream.
The *buf* parameter may change between invocations. Upon return from the callback, the *size* parameter must be modified to reflect the amount of data consumed from the buffer, and the buffer must not be freed even if all data is consumed. The function is expected to consume data in a linear (first in, first out) fashion. Unconsumed data is present at the beginning of the buffer at the next invocation of the callback. The *size* parameter will include the size of the unconsumed data.

Both callbacks' return values are expected to be zero. If unsuccessful, a positive value will be placed into the **errno** global variable (with the accompanying indication of an error return from the kernel service). If the return value of a callback is less than 0, end-of-file will be signaled to the user, and the return value will be treated as its unary negation (For example, -1 will be treated like 0). In this case, no further callbacks will be made for this stream.

The **tuning_register_bint32**, **tuning_register_buint32**, **tuning_register_bint64**, and **tuning_register_buint64** kernel services are used to add a file at the location specified by the *path* parameter that, when read from, will return the ASCII value of the integer variable pointed to by the *variable* parameter. When written to, this file will set the integer variable to the value whose ASCII value was written, unless that value does not satisfy the relation low <= value < high. In this case, the integer variable is not modified, and an error is returned to the user through an error return of the kernel service during which the invalid attempt is detected (probably either **write** or **close**).

The **tuning_register_b*x** functions operate similarly to their non-**x** variants, but they use a pair of callbacks to retrieve (*rfunc*) and set (*wfunc*) the variable. The callback is passed the value (if setting) and the context parameter. This permits more complex operations on read/write, such as serialization and memory allocation and deallocation.

The **tuning_get_context** kernel service returns the *context* of the registration function used to create the **tinode_t** structure referred to by the *argument* parameter.

The **tuning_register** kernel service is the basic interface by which a file can be added to the **/proc/sys** directory hierarchy. This function is not exported to kernel extensions, and its direct use in the kernel is strongly discouraged. The *path* parameter contains the path relative to the **/proc/sys** root at which the file should appear. Intermediate path components are automatically created. The *mode* parameter contains the UNIX permissions and the type of the file to be created (as per the **st_mode** field of the **stat** struct). If the file type is not specified, it is assumed to be **S_IFREG**. In most cases this parameter will be 0644 or 0600. The *vnops* parameter is used to dispatch all operations on the file.

The **tuning_deregister** kernel service is used to remove a file from the **/proc/sys** directory hierarchy. It is exported to kernel extensions. It should only be used when a specific file's implementation is no longer available. The *t* parameter is a **tinode_t** structure as returned by **tuning_register**. If the file is currently open, any further access to it after this call returns **ESTALE**.

Parameters

Item	Description
mode	Is set to either **TH_EOF** if no further data is expected from the user for this change, or **TH_MORE** if further data is expected.
size	Contains the size of the data passed in the buffer.
buf	Points to the buffer.
context	Points to the context passed to the registration function.
path	Specifies the location of the file to be added.
readfunc	Behaves as a producer function.
rfunc	Retrieves the variable.
wfunc	Sets the variable.
writefunc	Behaves as a consumer function.
variable	Specifies the variable.
high	Specifies the maximum value that the *variable* parameter can contain.
low	Specifies the minimum value that the *variable* parameter can contain.
t	A **tinode_t** structure as returned by **tuning_register**.

Return Values

Upon successful completion, the **tuning_register** kernel service returns the newly created **tinode_t** structure. If unsuccessful, a NULL value is returned.

Examples

A user of this interface might include the following line in their initialization routine:

```
tuning_var = tuning_register_buint64
("fs/jfs2/max_readahead", 0644 &j2_max_read_ahead, 0, 1024);
```

In this example *tuning_var* is a global variable of type **tinode_t ***. This causes the **fs** and **fs/jfs2** directories to be created, and a file (pipe) to be created as **fs/jfs2/max_readahead**. The file returns the value of **j2_max_readahead** in ASCII when read. The variable is read at the time of the first read. A write would set the value of the variable, but only at the time of either the first newline being written or a **close** function being performed. In order to write the variable after reading it, one must close the file and reopen it for write. This file is not seekable.

u

The following kernel services begin with the with the letter u.

ue_proc_check Kernel Service

Purpose

Determines if a process is critical to the system.

Syntax

```
int ue_proc_check (pid)
pid_t pid;
```

Description

The **ue_proc_check** kernel service determines if a particular process is critical to the system. A critical process is either a kernel process or a process registered as critical by the **ue_proc_register** system call. A process that is critical will cause the system to terminate if that process has an unrecoverable hardware error associated with the process. Unrecoverable hardware errors associated with a process are determined by the kernel machine check handler on systems that support UE-Gard error processing.

The **ue_proc_check** kernel service should be called only while executing in kernel mode in the user process.

Parameters

Item	Description
pid	Specifies the process' ID to be checked as critical.

Execution Environment

The **ue_proc_check** kernel service can be called from the interrupt environment only.

Return Values

Item	Description
0	Indicates that the *pid* is not critical.
EINVAL	Indicates that the *pid* is critical.
-1	Indicates that the *pid* parameter is not valid or the process no longer exists.

Related reference:
"ue_proc_register Subroutine"

ue_proc_register Subroutine

Purpose

Registers a process as critical to the system.

Syntax

```
int ue_proc_register (pid, argument)
pid_t pid;
int argument;
```

Description

The **ue_proc_register** system call registers a particular process as critical to the system. A process that is critical will cause the system to terminate if that process has an unrecoverable hardware error associated with the process. Unrecoverable hardware errors associated with a process are determined by the kernel machine check handler on systems that support UE-Gard error processing.

An execed process from a critical process must register itself to be critical. A fork from a process inherits the critical registration unless the argument is set to **NONCRITFORK**.

If the value of the *pid* parameter is equal to (**pid_t**) 0, the subroutine is registering the calling process.

The **ue_proc_register** system call should be called only while executing with root authority in the user process.

Parameters

Item	Description
pid	Specifies the process' ID to be registered critical.
argument	Defined in the **sys/proc.h** header file. Can be the following value: **NONCRITFORK** The *pid* forks are not critical.

Execution Environment

The **ue_proc_register** system call can be called from the process environment only.

Return Values

Item	Description
0	Indicates successful completion.
EINVAL	Indicates that the *pid* parameter is not valid or the process no longer exists.
EACCES	Indicates that the caller does not have sufficient authority to alter the *pid* registration.

Related reference:

"ue_proc_unregister Subroutine"

ue_proc_unregister Subroutine

Purpose

Unregisters a process from being critical to the system.

Syntax

```
int ue_proc_register (pid)
pid_t pid;
```

Description

The **ue_proc_unregister** system call unregisters a particular process as being no longer critical to the system. A process that has been previously registered critical will cause the system to terminate if that process has an unrecoverable hardware error associated with the process. Unrecoverable hardware errors associated with a process are determined by the kernel machine check handler on systems that support UE-Gard error processing.

If the value of the *pid* parameter is equal to (**pid_t**) 0, the subroutine is unregistering the calling process.

The **ue_proc_unregister** service should be called only while executing with root authority in the user process.

Parameters

Item	Description
pid	Specifies the process' ID to be unregistered.

Execution Environment

The **ue_proc_unregister** system call can be called from the process environment only.

Return Values

Item	Description
0	Indicates successful completion.
EINVAL	Indicates that the *pid* parameter is not valid or the process no longer exists.
EACCES	Indicates that the caller does not have sufficient authority to alter the *pid* registration.

Related reference:

"ue_proc_register Subroutine" on page 504

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

Item	Description
exp	Points to an exception handler structure. This structure must be pinned and is used for registering user-mode process exception handlers. The **uexcepth** structure is defined in the **/usr/include/sys/except.h** file.

Description

The **uexadd** kernel service is typically used to install a systemwide exception handler to catch exceptions occurring during execution of a process in user mode. The **uexadd** kernel service adds the exception handler structure specified by the *exp* parameter, to the chain of exception handlers to be called if an exception occurs while a process is executing in user mode. The last exception handler registered is the first exception handler called for a user-mode exception.

The **uexcepth** structure has:

- A chain element used by the kernel to chain the registered user exception handlers.
- A function pointer defining the entry point of the exception handler being added.

Additional exception handler-dependent information can be added to the end of the structure, but must be pinned.

> **Attention:** The **uexcepth** structure must be pinned when the **uexadd** kernel service is called. It must remain pinned and unmodified until after the call to the **uexdel** kernel service to delete the specified exception handler. Otherwise, the system may crash.

Execution Environment

The **uexadd** kernel service can be called from the process environment only.

Return Values

The **uexadd** kernel service has no return values.

Related reference:

"uexdel Kernel Service" on page 509

"User-Mode Exception Handler for the uexadd Kernel Service"

Related information:

User-Mode Exception Handling

Kernel Extension and Device Driver Management Services

User-Mode Exception Handler for the uexadd Kernel Service

Purpose

Handles exceptions that occur while a kernel thread is executing in user mode.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/except.h>

int func (exp, type, tid, mst)
struct excepth * exp;
int  type;
tid_t  tid;
struct kmstsave * mst;
```

Parameters

Item	Description
exp	Points to the **excepth** structure used to register this exception handler.
mst	Points to the current **kmstsave** area for the process. This pointer can be used to access the **kmstsave** area to obtain additional information about the exception.

Item	Description
tid	Specifies the thread ID of the kernel thread that was executing at the time of the exception.
type	Denotes the type of exception that has occurred. This type value is platform specific. Specific values are defined in the **/usr/include/sys/except.h** file.

Description

The user-mode exception handler (*exp*->**func**) is called for synchronous exceptions that are detected while a kernel thread is executing in user mode. The kernel exception handler saves exception information in the **kmstsave** area of the structure. For user-mode exceptions, it calls the first exception handler found on the user exception handler list. The exception handler executes in an interrupt environment at the priority level of either **INTPAGER** or **INTIODONE**.

If the registered exception handler returns a return code indicating that the exception was handled, the kernel exits from the exception handler without calling additional exception handlers from the list. If the exception handler returns a return code indicating that the exception was not handled, the kernel invokes the next exception handler on the list. The last exception handler in the list is the default handler. This is typically signalling the thread.

The kernel exception handler must not page fault. It should also register an exception handler using the **setjmpx** kernel service if any exception-handling activity can result in an exception. This is important particularly if the exception handler is handling the I/O. If the exception handler did not handle the exception, the return code should be set to the **EXCEPT_NOT_HANDLED** value for user-mode exception handling.

Execution Environment

The user-mode exception handler for the **uexadd** kernel service is called in the interrupt environment at the **INTPAGER** or **INTIODONE** priority level.

Return Values

Item	Description
EXCEPT_HANDLED	Indicates that the exception was successfully handled.
EXCEPT_NOT_HANDLED	Indicates that the exception was not handled.

Related reference:

"uexadd Kernel Service" on page 505

Related information:

User-Mode Exception Handling

Kernel Extension and Device Driver Management Kernel Services

uexblock Kernel Service

Purpose

Makes the currently active kernel thread nonrunnable when called from a user-mode exception handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/except.h>

void uexblock ( tid)
tid_t *tid;
```

Parameter

Item	Description
tid	Specifies the thread ID of the currently active kernel thread to be put into a wait state.

Description

The **uexblock** kernel service puts the currently active kernel thread specified by the *tid* parameter into a wait state until the **uexclear** kernel service is used to make the thread runnable again. If the **uexblock** kernel service is called from the process environment, the *tid* parameter must specify the current active thread; otherwise the system will crash with a kernel panic.

The **uexblock** kernel service can be used to lazily control user-mode threads access to a shared serially usable resource. Multiple threads can use a serially used resource, but only one process at a time. When a thread attempts to but cannot access the resource, a user-mode exception can be set up to occur. This gives control to an exception handler registered by the **uexadd** kernel service. This exception handler can then block the thread using the **uexblock** kernel service until the resource is made available. At this time, the **uexclear** kernel service can be used to make the blocked thread runnable.

Execution Environment

The **uexblock** kernel service can be called from either the process or interrupt environment.

Return Values

The **uexblock** service has no return values.

Related reference:

"uexclear Kernel Service" on page 509

Related information:

User-Mode Exception Handling

Kernel Extension and Device Driver Management Services

uexclear Kernel Service

Purpose

Makes a kernel thread blocked by the **uexblock** service runnable again.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/except.h>

void uexclear ( tid)
tid_t *tid;
```

Parameter

Item	Description
tid	Specifies the thread ID of the previously blocked kernel thread to be put into a run state.

Description

The **uexclear** kernel service puts a kernel thread specified by the *tid* parameter back into a runnable state after it was made nonrunnable by the **uexblock** kernel service. A thread that has been sent a **SIGSTOP** stop signal is made runnable again when it receives the **SIGCONT** continuation signal.

The **uexclear** kernel service can be used to lazily control user-mode thread access to a shared serially usable resource. A serially used resource is usable by more than one thread, but only by one at a time. When a thread attempts to access the resource but does not have access, a user-mode exception can be setup to occur.

This setup gives control to an exception handler registered by the **uexadd** kernel service. Using the **uexblock** kernel service, this exception handler can then block the thread until the resource is later made available. At that time, the **uexclear** service can be used to make the blocked thread runnable.

Execution Environment

The **uexclear** kernel service can be called from either the process or interrupt environment.

Return Values

The **uexclear** service has no return values.

Related reference:

"uexblock Kernel Service" on page 508

Related information:

User-Mode Exception Handling

Kernel Extension and Device Driver Management Services

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

Item	Description
exp	Points to the exception handler structure used to add the exception handler with the **uexadd** kernel service.

Description

The **uexdel** kernel service removes a user-mode exception handler from the systemwide list of exception handlers maintained by the kernel's exception handler.

The **uexdel** kernel service removes the exception handler structure specified by the *exp* parameter from the chain of exception handlers to be called if an exception occurs while a process is executing in user mode. Once the **uexdel** kernel service has completed, the specified exception handler is no longer called. In addition, the **uexcepth** structure can be modified, freed, or unpinned.

Execution Environment

The **uexdel** kernel service can be called from the process environment only.

Return Values

The **uexdel** kernel service has no return values.

Related reference:

"uexadd Kernel Service" on page 505

Related information:

User-Mode Exception Handling

Kernel Extension and Device Driver Management Services

ufdcreate Kernel Service

Purpose

Allocates and initializes a file descriptor.

Syntax

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/file.h>

int ufdcreate (flags, ops, datap, type, fdp, cnp)

int  flags;
struct fileops * ops;
void * datap;
short  type;
int * fdp;
 struct ucred *crp;
```

Parameters

Item	Description
flags	Specifies the flags to save in a **file** structure. The **file** structure is defined in the **sys/file.h** file. If a **read** or **write** subroutine is called with the file descriptor returened by this routine, the **FREAD** and **FWRITE** flags must be set appropriately. Valid flags are defined in the **fcntl.h** file.
ops	Points to the list of subsystem-supplied routines to call for the file system operations: read/write, ioctl, select, fstat, and close. The **fileops** structure is defined in the **sys/file.h** file. See "File Operations" for more information.
datap	Points to type-dependent structures. The system saves this pointer in the **file** structure. As a result, the pointer is available to the file operations when they are called.
type	Specifies the unique type value for the **file** structure. Valid types are listed in the **sys/file.h** file.
fdp	Points to an integer field where the file descriptor is stored on successful return.
crp	Points to a credentials structure. This pointer is saved in the file struct for use in subsequent operations. It must be a valid **ucred** struct. The **crref()** kernel service can be used to obtain a **ucred** struct.

Description

The **ufdcreate** kernel service provides a file interface to kernel extensions. Kernel extensions use this service to create a file descriptor and file structure pair. Also, this service allows kernel extensions to provide their own file descriptor-based system calls, enabling read/write, ioctl, select, fstat, and close operations on objects outside the file system. The **ufdcreate** kernel services does not require the extension to understand or conform to the synchronization requirements of the logical file system (LFS).

The **ufdcreate** kernel service provides a file descriptor to the caller and creates the underlying file structure. The caller must include pointers to subsystem-supplied routines for the read/write, ioctl, select, fstat, and close operations. If any of the operations are not needed by the calling subsystem, then the caller must provide a pointer to an appropriate **errno** value. Typically, the **EOPNOTSUPP** value is used for this purpose. See "File Operations" for information about the requirements for the subsystem-supplied routines.

Removing a File Descriptor

There is no corresponding operation to remove a file descriptor (and the attendant structures) created by the **ufdcreate** kernel service. To remove a file descriptor, use a call to the **close** subroutine. The **close** subroutine can be called from a routine or from within the kernel or kernel extension. If the close is not called, the file is closed when the process exits.

Once a call is made to the **ufdcreate** kernel service, the file descriptor is considered open before the call to the service returns. When a **close** or **exit** subroutine is called, the close file operation specified on the call to the **ufdcreate** interface is called.

File Operations

The **ufdcreate** kernel service allows kernel extensions to provide their own file descriptor-based system calls, enabling read/write, ioctl, select, fstat, and close operations on objects outside the file system. The **fileops** structure defined in the **sys/file.h** file provides interfaces for these routines.

read/write Requirements

The read/write operation manages input and output to the object specified by the *fp* parameter. The actions taken by this operation are dependent on the object type. The syntax for the operation is as follows:

```
#include <sys/types.h>
#include <sys/uio.h>

int (*fo_rw) (fp, rw, uiop, ext)
```

```
struct file *fp;
enum uio_rw rw;
struct uio *uiop;
int ext;
```

The parameters have the following values:

Value	Description
fp	Points to the **file** structure. This structure corresponds to the file descriptor used on the **read** or **write** subroutine.
rw	Contains a **UIO_READ** value for a read operation or **UIO_WRITE** value for a write operation.
uiop	Points to a **uio** structure. This structure describes the location and size information for the input and output requested. The **uio** structure is defined in the **uio.h** file.
ext	Specifies subsystem-dependent information. If the **readx** or **writex** subroutine is used, the value passed by the operation is passed through to this subroutine. Otherwise, the value is 0.

If successful, the **fo_rw** operation returns a value of 0. A nonzero return value should be programmed to indicate an error. See the **sys/errno.h** file for a list of possible values.

Note: On successful return, the `uiop->uio_resid` field must be updated to include the number of bytes of data actually transferred.

ioctl Requirements

The ioctl operation provides object-dependent special command processing. The **ioctl** subroutine performs a variety of control operations on the object associated with the specified open **file** structure. This subroutine is typically used with character or block special files and returns an error for ordinary files.

The control operation provided by the ioctl operation is specific to the object being addressed, as are the data type and contents of the *arg* parameter.

The syntax for the ioctl operation is as follows:

```
#include <sys/types.h>
#include <sys/ioctl.h>

int (*fo_ioctl) (fp, cmd, arg, ext, kflag)

struct file *fp;
int cmd, ext, kflag;
caddr_t arg;
```

The parameters have the following values:

Value	Description
fp	Points to the **file** structure. This structure corresponds to the file descriptor used by the **ioctl** subroutine.
cmd	Defines the specific request to be acted upon by this routine.
arg	Contains data that is dependent on the *cmd* parameter.
ext	Specifies subsystem-specific information. If the **ioctlx** subroutine is used, the value passed by the application is passed through to this subroutine. Otherwise, the value is 0.
kflag	Determines where the call is made from. The *kflag* parameter has the value **FKERNEL** (from the **fcntl.h** file) if this routine is called through the **fp_ioctl** interface. Otherwise, its value is 0.

If successful, the **fo_ioctl** operation returns a value of 0. For errors, the **fo_ioctl** operation should return a nonzero return value to indicate an error. Refer to the **sys/errno.h** file for the list of possible values.

select Requirements

The select operation performs a select operation on the object specified by the *fp* parameter. The syntax for this operation is as follows:

```
#include <sys/types.h>
int (*fo_select) (fp, corl, reqevents, rtneventsp, notify)
struct file *fp;
int corl;
ushort reqevents, *rtneventsp;
void (notify) ();
```

The parameters have the following values:

Value	Description
fp	Points to the **file** structure. This structure corresponds to the file descriptor used by the **select** subroutine.
corl	Specifies the ID used for correlation in the **selnotify** kernel service.
reqevents	Identifies the events to check. The poll and select functions define three standard event flags and one informational flag. The **sys/poll.h** file details the event bit definition. See the **fp_select** kernel service for information about the possible flags.
rtneventsp	Indicates the returned events pointer. This parameter, passed by reference, indicates the events that are true at the current time. The returned event bits include the request events and an error event indicator.
notify	Points to a routine to call when the specified object invokes the **selnotify** kernel service for an outstanding asynchronous select or poll event request. If no routine is to be called, this parameter must be null.

If successful, the **fo_select** operation returns a value of 0. This operation should return a nonzero return value to indicate an error. Refer to the **sys/errno.h** file for the list of possible values.

fstat Requirements

The fstat operation fills in an **attribute** structure. Depending on the object type specified by the *fp* parameter, many fields in the structure may not be applicable. The value passed back from this operation is dependent upon both the object type and what any routine that understands the type is expecting. The syntax for this operation is as follows:

```
#include <sys/types.h>
int (*fo_fstat) (fp, sbp)
struct file *fp;
struct stat *sbp;
```

The parameters have the following values:

Value	Description
fp	Points to the **file** structure. This structure corresponds to the file descriptor used by the **stat** subroutine.
sbp	Points to the **stat** structure to be filled in by this operation. The address supplied is in kernel space.

If successful, the **fo_fstat** operation returns a value of 0. A nonzero return value should be programmed to indicate an error. Refer to the **sys/errno.h** file for the list of possible values.

close Requirements

The close operation invalidates routine access to objects specified by the *fp* parameter and releases any data associated with that access. This operation is called from the **close** subroutine code when the **file** structure use count is decremented to 0. For example, if there are multiple accesses to an object (created by the **dup**, **fork**, or other subsystem-specific operation), the **close** subroutine calls the close operation when it determines that there is no remaining access through the **file** structure being closed.

A file descriptor is considered open once a file descriptor and **file** structure have been set up by the LFS. The close file operation is called whenever a close or exit is specified. As a result, the close operation must be able to close an object that is not fully open, depending on what the caller did before the **file** structure was initialized.

The syntax for the close operation is as follows:

```
#include <sys/file.h>

int (*fo_close) (fp)
struct file *fp;
```

The parameter is:

Item	Description
fp	Points to the **file** structure. This structure corresponds to the file descriptor used by the **close** subroutine.

If successful, the **fo_close** operation returns a value of 0. This operation should return a nonzero return value to indicate an error. Refer to the **sys/errno.h** file for the list of possible values.

Execution Environment

The **ufdcreate** kernel service can be called from the process environment only.

Return Values

If the **ufdcreate** kernel service succeeds, it returns a value of 0. If the kernel service fails, it returns a nonzero value and sets the **errno** global variable.

Error Codes

The **ufdcreate** kernel service fails if one or more of the following errors occur:

Error	Description
EINVAL	The *ops* parameter is null, or the **fileops** structure does not have entries for for every operation.
EMFILE	All file descriptors for the process have already been allocated.
ENFILE	The system file table is full.

Related reference:

"selnotify Kernel Service" on page 462

Related information:

close subroutine

exit, atexit, or _exit

Logical File System Kernel Services

ufdgetf Kernel Service

Purpose

Returns a pointer to a file structure associated with a file descriptor.

Syntax

```
#include <sys/file.h>

int ufdgetf( fd,  fpp)
int fd;
struct file **fpp;
```

Parameters

Item	Description
fd	Identifies the file descriptor. The descriptor must be for an open file.
fpp	Points to a location to store the file pointer.

Description

The **ufdgetf** kernel service returns a pointer to a file structure associated with a file descriptor. The calling routine must have a use count on the file descriptor. To obtain a use count on the file descriptor, the caller must first call the **ufdhold** kernel service.

Execution Environment

The **ufdget** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates successful completion.
EBADF	Indicates that the *fd* parameter is not a file descriptor for an open file.

Related reference:
"ufdhold and ufdrele Kernel Service"

ufdhold and ufdrele Kernel Service

Purpose

Increment or decrement a file descriptor reference count.

Syntax

```
int ufdhold( fd)
int fd;
```

```
int ufdrele(fd)
int fd;
```

Parameter

Item	Description
fd	Identifies the file descriptor.

Description

> **Attention:** It is extremely important that the calls to **ufdhold** and **ufdrele** kernel service are balanced. If a file descriptor is held more times than it is released, the **close** subroutine on the descriptor never completes. The process hangs and cannot be killed. If the descriptor is released more times than it is held, the system panics.

The **ufdhold** and **ufdrele** kernel services increment and decrement a file-descriptor reference count. Together, these kernel services maintain the file descriptor reference count. The **ufdhold** kernel service increments the count. The **ufdrele** kernel service decrements the count.

These subroutines are supported for kernel extensions that provide their own file-descriptor-based system calls. This support is required for synchronization with the **close** subroutine.

When a thread is executing a file-descriptor-based system call, it is necessary that the logical file system (LFS) be aware of it. The LFS uses the count in the file descriptor to monitor the number of system calls currently using any particular file descriptor. To keep the count accurately, any thread using the file descriptor must increment the count before performing any operation and decrement the count when all activity using the file descriptor is completed for that system call.

Execution Environment

These kernel services can be called from the process environment only.

Return Values

Item	Description
0	Indicates successful completion.
EBADF	Indicates that the *fd* parameter is not a file descriptor for an open file.

Related reference:

"ufdgetf Kernel Service" on page 514

Related information:

close subroutine

uiomove Kernel Service

Purpose

Moves a block of data between kernel space and a space defined by a **uio** structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int uiomove ( cp, n, rw, uiop)
caddr_t cp;
int n;
uio_rw rw;
struct uio *uiop;
```

Parameters

Item	Description
cp	Specifies the address in kernel memory to or from which data is moved.
n	Specifies the number of bytes to move.
rw	Indicates the direction of the move: **UIO_READ** Copies data from kernel space to space described by the **uio** structure. **UIO_WRITE** Copies data from space described by the **uio** structure to kernel space.
uiop	Points to a **uio** structure describing the buffer used in the data transfer.

Description

The **uiomove** kernel service moves the specified number of bytes of data between kernel space and a space described by a **uio** structure. Device driver top halves, especially character device drivers, frequently use the **uiomove** service to transfer data into or out of a user area. The `uio_resid` and `uio_iovcnt` fields in the **uio** structure describing the data area must be greater than 0 or an error is returned.

The **uiomove** service moves the number of bytes of data specified by either the *n* or *uio_resid* parameter, whichever is less. If either the *n* or *uio_resid* parameter is 0, no data is moved. The `uio_segflg` field in the **uio** structure is used to indicate if the move is accessing a user- or kernel-data area, or if the caller requires cross-memory operations and has provided the required cross-memory descriptors. If a cross-memory operation is indicated, there must be a cross-memory descriptor in the **uio_xmem** array for each `iovec` element.

If the move is successful, the following fields in the **uio** structure are updated:

Field	**Description**
`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

Item	**Description**
0	Indicates successful completion.
ENOMEM	Indicates that there was no room in the buffer.
EIO	Indicates a permanent I/O error file space.
ENOSPC	Indicates insufficient disk space.
EFAULT	Indicates a user location that is not valid.

Related reference:

"uphysio Kernel Service" on page 523

"uio Structure" on page 638

Related information:

Memory Kernel Services

unlock_enable Kernel Service

Purpose

Unlocks a simple lock if necessary, and restores the interrupt priority.

Syntax

```
#include <sys/lock_def.h>

void unlock_enable ( int_pri,  lock_addr)
int int_pri;
simple_lock_t lock_addr;
```

Parameters

Item	Description
int_pri	Specifies the interrupt priority to restore. This must be set to the value returned by the corresponding call to the **disable_lock** kernel service.
lock_addr	Specifies the address of the lock word to unlock.

Description

The **unlock_enable** kernel service unlocks a simple lock if necessary, and restores the interrupt priority, in order to provide optimized thread-interrupt critical section protection for the system on which it is executing. On a multiprocessor system, calling the **unlock_enable** kernel service is equivalent to calling the **simple_unlock** and **i_enable** kernel services. On a uniprocessor system, the call to the **simple_unlock** service is not necessary, and is omitted. However, you should still pass the valid lock address which was used with the corresponding call to the **disable_lock** kernel service. Never pass a **NULL** lock address.

Execution Environment

The **unlock_enable** kernel service can be called from either the process or interrupt environment.

Return Values

The **unlock_enable** kernel service has no return values.

Related reference:

"disable_lock Kernel Service" on page 76

"simple_unlock Kernel Service" on page 475

Related information:

Understanding Locking

Understanding Interrupts

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

Item	Description
lock_word	Specifies the address of the lock word.

Description

Note: The **unlockl** kernel service is provided for compatibility only and should not be used in new code, which should instead use simple locks or complex locks.

The **unlockl** kernel service unlocks a conventional lock. Only the owner of a lock can unlock it. Once a lock is unlocked, the highest priority thread (if any) which is waiting for the lock is made runnable and may compete again for the lock. If there was at least one process waiting for the lock, the priority of the caller is recomputed. Preempting a System Call discusses how system calls can use locking kernel services when accessing global data.

The **lockl** and **unlockl** services do not maintain a nesting level count. A single call to the **unlockl** service unlocks the lock for the caller. The return code from the **lockl** service should be used to determine when to unlock the lock.

Note: The **unlockl** kernel service can be called with interrupts disabled, only if the event or lock word is pinned.

Execution Environment

The **unlockl** kernel service can be called from the process environment only.

Return Values

The **unlockl** service has no return values.

Example

A call to the **unlockl** service can be coded as follows:

```
int lock_ret;          /* return code from lockl() */
extern int lock_word;  /* lock word that is external
                          and was initialized to
                          LOCK_AVAIL */
...
/* get lock prior to using resource */
lock_ret = lockl(lock_word, LOCK_SHORT)
/* use resource for which lock was obtained */
...
/* release lock if this was not a nested use */
if ( lock_ret != LOCK_NEST )
   unlockl(lock_word);
```

Related reference:

"lockl Kernel Service" on page 342

Related information:

Understanding Locking

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

Item	Description
addr	Specifies the address of the first byte to unpin in the system (kernel) address space.
length	Specifies the number of bytes to unpin.

Description

The **unpin** kernel service decreases the pin count of each page in the address range. When the pin count is 0, the page is not pinned and can be paged out of real memory. Upon finding an unpinned page, the **unpin** service returns the **EINVAL** error code and leaves any remaining pinned pages still pinned.

The **unpin** service can only be called with addresses in the system (kernel) address space. The **xmemunpin** 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

Item	Description
0	Indicates successful completion.
EINVAL	Indicates that the value of the *length* parameter is negative or 0. Otherwise, the area of memory beginning at the byte specified by the *base* parameter and extending for the number of bytes specified by the *len* parameter is not defined. If neither cause is responsible, an unpinned page was specified.

Related reference:

"pin Kernel Service" on page 407

"xmemunpin Kernel Service" on page 605

Related information:

Understanding Execution Environments

Memory Kernel Services

unpincode Kernel Service

Purpose

Unpins the code and data associated with a loaded object module.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/pin.h>

int unpincode ( func)
int (*func) ( );
```

Parameter

Item	Description
func	Specifies an address used to determine the object module to be unpinned. The address is typically that of a function that is exported by this object module.

Description

The **unpincode** kernel service uses the **ltunpin** kernel service to decrement the pin count for the pages associated with the following items:

- Code associated with the object module
- Data area of the object module that contains the function specified by the *func* parameter

The loader entry for the module is used to determine the size of both the code and the data area.

Execution Environment

The **unpincode** kernel service can be called from the process environment only.

Return Values

Item	Description
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.

Related reference:

"unpin Kernel Service" on page 519

Related information:

Understanding Execution Environments

Memory Kernel Services

unregister_HA_handler Kernel Service

Purpose

Removes from the kernel the registration of a High Availability Event Handler.

Syntax

```
#include <sys/high_avail.h>

int register_HA_handler (ha_handler)
ha_handler_ext_t  * ha_handler;
```

Parameter

Item	Description
ha_handler	Specifies a pointer to a structure of the type **ha_handler_ext_t** defined in /**usr/include/sys/high_avail.h**. This structure must be identical to the one passed to **register_HA_handler** at the time of registration.

Description

The **unregister_HA_handler** kernel service cancels an unconfigured kernel extensions that have registered a high availability event handler, done by the **register_HA_handler** kernel service, so that the kernel extension can be unloaded.

Failure to do so may cause a system crash when a high availability event such as a processor deallocation is initiated due to some hardware fault.

Execution Environment

The **unregister_HA_handler** kernel service can be called from the process environment only.

An extension may register the same HAEH *N* times (*N* > 1). Although this is considered an incorrect behaviour, no error is reported. The given HAEH will be invoked *N* times for each HA event. This handler has to be unregistered as many times as it was registered.

Return Values

Item	Description
0	Indicates a successful operation.

A non-zero value indicates an error.

Related reference:

"register_HA_handler Kernel Service" on page 446

Related information:

RAS Kernel Services

untimeout Kernel Service

Attention: This service must not be used because it is not multi-processor safe. The base kernel timer and watchdog services must be used instead. See talloc and w_init for more information.

Purpose

Cancels a pending timer request.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

void untimeout ( func,  arg)
void (*func)();
caddr_t *arg;
```

Parameters

Item	Description
func	Specifies the function that is associated with the timer to be canceled.
arg	Specifies the function argument that is associated with the timer to be canceled.

Description

The **untimeout** kernel service is not part of the kernel. However, it is a compatibility service that is provided in the **libsys.a** library. To use the **untimeout** service, a kernel extension must be bound with the **libsys.a** library. The **untimeout** service, like the associated kernel libsys services **timeoutcf** and **timeout**, can be bound and used only in the pinned part of a kernel extension or the bottom half of a device driver because these services use interrupt disable for serialization.

The **untimeout** kernel service cancels a specific request that is made with the **timeout** service. The *func* and *arg* parameters must match the parameters that are used in the **timeout** kernel service request that is to be canceled.

Upon return, the specified timer request is canceled, if found. If no timer request matches the *func* and *arg* parameters, no operation is performed.

Execution Environment

The **untimeout** kernel service can be called from either the process or interrupt environment.

Return Values

The **untimeout** kernel service has no return values.

Related reference:

"timeout Kernel Service" on page 489

Related information:

Timer and Time-of-Day Kernel Services

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

Item	Description
uiop	Points to the **uio** structure describing the buffer of data to transfer using character-to-block I/O.
rw	Indicates either a read or write operation. A value of **B_READ** for this flag indicates a read operation. A value of **B_WRITE** for this flag indicates a write operation.
buf_cnt	Specifies the maximum number of **buf** structures to use when calling the strategy routine specified by the *strat* parameter. This parameter is used to indicate the maximum amount of concurrency the device can support and minimize the I/O redrive time. The value of the *buf_cnt* parameter can range from 1 to 64.
devno	Specifies the major and minor device numbers. With the **uphysio** service, this parameter specifies the device number to be placed in the **buf** structure before calling the strategy routine specified by the *strat* parameter.
strat	Represents the function pointer to the **ddstrategy** routine for the device.
mincnt	Represents the function pointer to a routine used to reduce the data transfer size specified in the **buf** structure, as required by the device before the strategy routine is started. The routine can also be used to update extended parameter information in the **buf** structure before the information is passed to the strategy routine.
minparms	Points to parameters to be used by the *mincnt* parameter.

Description

The **uphysio** kernel service performs character I/O for a block device. The **uphysio** service attempts to send to the specified strategy routine the number of **buf** headers specified by the *buf_cnt* parameter. These **buf** structures are constructed with data from the **uio** structure specified by the *uiop* parameter.

The **uphysio** service initially transfers data area descriptions from each `iovec` element found in the **uio** structure into individual **buf** headers. These headers are later sent to the strategy routine. The **uphysio** kernel service tries to process as many data areas as the number of **buf** headers permits. It then invokes the strategy routine with the list of **buf** headers.

Preparing Individual buf Headers

The routine specified by the *mincnt* parameter is called before the **buf** header, built from an `iovec` element, is added to the list of **buf** headers to be sent to the strategy routine. The *mincnt* parameter is passed a pointer to the **buf** header along with the *minparms* pointer. This arrangement allows the *mincnt* parameter to tailor the length of the data transfer described by the **buf** header as required by the device performing the I/O. The *mincnt* parameter can also optionally modify certain device-dependent fields in the **buf** header.

When the *mincnt* parameter returns with no error, an attempt is made to pin the data buffer described by the **buf** header. If the pin operation fails due to insufficient memory, the data area described by the **buf** header is reduced by half. The **buf** header is again passed to the *mincnt* parameter for modification before trying to pin the reduced data area.

This process of downsizing the transfer specified by the **buf** header is repeated until one of the three following conditions occurs:

- The pin operation succeeds.
- The *mincnt* parameter indicates an error.
- The data area size is reduced to 0.

When insufficient memory indicates a failed pin operation, the number of **buf** headers used for the remainder of the operation is reduced to 1. This is because trying to pin multiple data areas simultaneously under these conditions is not desirable.

If the user has not already obtained cross-memory descriptors, further processing is required. (The uio_segflg field in the **uio** structure indicates whether the user has already initialized the cross-memory descriptors. The **usr/include/sys/uio.h** file contains information on possible values for this flag.)

When the data area described by the **buf** header has been successfully pinned, the **uphysio** service verifies user access authority for the data area. It also obtains a cross-memory descriptor to allow the device driver interrupt handler limited access to the data area.

Calling the Strategy Routine

After the **uphysio** kernel service obtains a cross-memory descriptor to allow the device driver interrupt handler limited access to the data area, the **buf** header is then put on a list of **buf** headers to be sent to the strategy routine specified by the *strat* parameter.

The strategy routine specified by the *strat* parameter is called with the list of **buf** headers when:
- The list reaches the number of **buf** structures specified by the *buf_cnt* parameter.
- The data area described by the **uio** structure has been completely described by **buf** headers.

The **buf** headers in the list are chained together using the av_back and av_forw fields before they are sent to the strategy routine.

Waiting for buf Header Completion

When all available **buf** headers have been sent to the strategy routine, the **uphysio** service waits for one or more of the **buf** headers to be marked complete. The **IODONE** handler is used to wake up the **uphysio** service when it is waiting for completed **buf** headers from the strategy routine.

When the **uphysio** service is notified of a completed **buf** header, the associated data buffer is unpinned and the cross-memory descriptor is freed. (However, the cross-memory descriptor is freed only if the user had not already obtained it.) An error is detected on the data transfer under the following conditions:
- The completed **buf** header has a nonzero b_resid field.
- The b_flags field has the **B_ERROR** flag set.

When an error is detected by the **uphysio** service, no new **buf** headers are sent to the strategy routine.

The **uphysio** service waits for any **buf** headers already sent to the strategy routine to be completed and then returns an error code to the caller. If no errors are detected, the **buf** header and any other completed **buf** headers are again used to send more data transfer requests to the strategy routine as they become available. This process continues until all data described in the **uio** structure has been transferred or until an error has been detected.

The **uphysio** service returns to the caller when:
- All **buf** headers have been marked complete by the strategy routine.
- All data specified by the **uio** structure has been transferred.

The **uphysio** service also returns an error code to the caller if an error is detected.

Error Detection by the uphysio Kernel Service

When it detects an error, the **uphysio** kernel service reports the error that was detected closest to the start of the data area described by the **uio** structure. No additional **buf** headers are sent to the strategy routine. The **uphysio** kernel service waits for all **buf** headers sent to the strategy routine to be marked complete.

However, additional **buf** headers may have been sent to the strategy routine between these two events:

- After the strategy routine detects the error.
- Before the **uphysio** service is notified of the error condition in the completed **buf** header.

When errors occur, various fields in the returned **uio** structure may or may not reflect the error. The `uio_iov` and `uio_iovcnt` fields are not updated and contain their original values.

The `uio_resid` and `uio_offset` fields in the returned **uio** structure indicate the number of bytes transferred by the strategy routine according to the sum of all (the `b_bcount` field minus the `b_resid` fields) fields in the **buf** headers processed by the strategy routine. These headers include the **buf** header indicating the error nearest the start of the data area described by the original **uio** structure. Any data counts in **buf** headers completed after the detection of the error are not reflected in the returned **uio** structure.

Execution Environment

The **uphysio** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates successful completion.
ENOMEM	Indicates that no memory is available for the required **buf** headers.
EAGAIN	Indicates that the operation fails due to a temporary insufficient resource condition.
EFAULT	Indicates that the `uio_segflg` field indicated user space and that the user does not have authority to access the buffer.
EIO or the b_error field in a buf header	Indicates an I/O error in a **buf** header processed by the strategy routine.
Return code from the mincnt parameter	Indicates that the return code from the *mincnt* parameter if the routine returned with a nonzero return code.

Related reference:

"uphysio Kernel Service mincnt Routine"

"buf Structure" on page 614

"uio Structure" on page 638

uphysio Kernel Service mincnt Routine

Purpose

Tailors a **buf** data transfer request to device-dependent requirements.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/buf.h>

int mincnt ( bp,  minparms)
struct buf *bp;
void *minparms;
```

Parameters

Item	Description
bp	Points to the **buf** structure to be tailored.
minparms	Points to parameters.

Description

Only the following fields in the **buf** header sent to the routine specified by the **uphysio** kernel service *mincnt* parameter can be modified by that routine:

- `b_bcount`
- `b_work`
- `b_options`

The *mincnt* parameter cannot modify any other fields without the risk of error. If the *mincnt* parameter determines that the **buf** header cannot be supported by the target device, the routine should return a nonzero return code. This stops the **buf** header and any additional **buf** headers from being sent to the **ddstrategy** routine.

The **uphysio** kernel service waits for all **buf** headers already sent to the strategy routine to complete and then returns with the return code from the *mincnt* parameter.

Related reference:

"uphysio Kernel Service" on page 523

uprintf Kernel Service

Purpose

Submits a request to print a message to the controlling terminal of a process.

Syntax

```
#include <sys/uprintf.h>

int uprintf ( Format [,
Value, ...])
char *Format;
```

Parameters

Item — **Description**

Format
Specifies a character string containing either or both of two types of objects:

- Plain characters, which are copied to the message output stream.
- Conversion specifications, each of which causes 0 or more items to be retrieved from the *Value* parameter list. Each conversion specification consists of a % (percent sign) followed by a character that indicates the type of conversion to be applied:

  % Performs no conversion. Prints %.

  d, i Accepts an integer *Value* and converts it to signed decimal notation.

  u Accepts an integer *Value* and converts it to unsigned decimal notation.

  o Accepts an integer *Value* and converts it to unsigned octal notation.

  x Accepts an integer *Value* and converts it to unsigned hexadecimal notation.

  s Accepts a *Value* as a string (character pointer), and characters from the string are printed until a \ 0 (null character) is encountered. *Value* must be non-null and the maximum length of the string is limited to **UP_MAXSTR** characters.

  Field width or precision conversion specifications are not supported.

  The following constants are defined in the **/usr/include/sys/uprintf.h** file:

  – **UP_MAXSTR**

  – **UP_MAXARGS**

  – **UP_MAXCAT**

  – **UP_MAXMSG**

  The *Format* string may contain from 0 to the number of conversion specifications specified by the **UP_MAXARGS** constant. The maximum length of the *Format* string is the number of characters specified by the **UP_MAXSTR** constant. *Format* must be non-null.

  The maximum length of the constructed kernel message is limited to the number of characters specified by the **UP_MAXMSG** constant. Messages larger then the number of characters specified by the **UP_MAXMSG** constant are discarded.

Value
Specifies, as an array, the value to be converted. The number, type, and order of items in the *Value* parameter list should match the conversion specifications within the *Format* string.

Description

The **uprintf** kernel service submits a kernel message request. Once the request has been successfully submitted, the **uprintfd** daemon constructs the message based on the *Format* and *Value* parameters of the request. The **uprintfd** daemon then writes the message to the process' controlling terminal.

Execution Environment

The **uprintf** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
ENOMEM	Indicates that memory is not available to buffer the request.
ENODEV	Indicates that a controlling terminal does not exist for the process.
ESRCH	Indicates that the **uprintfd** daemon is not active. No requests may be submitted.
EINVAL	Indicates that a string *Value* string pointer is null or the string *Value* parameter is greater than the number of characters specified by the **UP_MAXSTR** constant.
EINVAL	Indicates one of the following: • *Format* string pointer is null. • Number of characters in the *Format* string is greater than the number specified by the **UP_MAXSTR** constant. • Number of conversion specifications contained within the *Format* string is greater than the number specified by the **UP_MAXARGS** constant.

Related reference:

"NLuprintf Kernel Service" on page 385

Related information:

uprintfd command

Process and Exception Management Kernel Services

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

Item	Description
c	Specifies a character to be written to the buffer.
uiop	Points to a **uio** structure describing the buffer in which to place a character.

Description

The **ureadc** kernel service writes a character to a buffer described by a **uio** structure. Device driver top half routines, especially character device drivers, frequently use the **ureadc** kernel service to transfer data into a user area.

The `uio_resid` and `uio_iovcnt` fields in the **uio** structure describing the data area must be greater than 0. If these fields are not greater than 0, an error is returned. The `uio_segflg` field in the **uio** structure is used to indicate whether the data is being written to a user- or kernel-data area. It is also used to indicate if the caller requires cross-memory operations and has provided the required cross-memory descriptors. The values for the flag are defined in the **/usr/include/sys/uio.h** file.

If the data is successfully written, the following fields in the **uio** structure are updated:

Field	Description
`uio_iov`	Specifies the address of current `iovec` element to use.
`uio_xmem`	Specifies the address of current `xmem` element to use (used for cross-memory copy).
`uio_iovcnt`	Specifies the number of remaining `iovec` elements.
`uio_iovdcnt`	Specifies the number of `iovec` elements already processed.
`uio_offset`	Specifies the character offset on the device from which data is read.
`uio_resid`	Specifies the total number of characters remaining in the data area described by the `uio` structure.
`iov_base`	Specifies the address of the next available character in the data area described by the current `iovec` element.
`iov_len`	Specifies the length of remaining data area in the buffer described by the current `iovec` element.

Execution Environment

The **ureadc** kernel service can be called from the process environment only.

Return Values

Item	Description
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.

Related reference:

"uiomove Kernel Service" on page 516

"uwritec Kernel Service"

Related information:

Memory Kernel Services

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

Item	Description
uiop	Points to a **uio** structure describing the buffer from which to read a character.

Description

The **uwritec** kernel service reads a character from a buffer described by a **uio** structure. Device driver top half routines, especially character device drivers, frequently use the **uwritec** kernel service to transfer data out of a user area. The `uio_resid` and `uio_iovcnt` fields in the **uio** structure must be greater than 0 or an error is returned.

The `uio_segflg` field in the **uio** structure indicates whether the data is being read out of a user- or kernel-data area. This field also indicates whether the caller requires cross-memory operations and has provided the required cross-memory descriptors. The values for this flag are defined in the **/usr/include/sys/uio.h** file.

If the data is successfully read, the following fields in the **uio** structure are updated:

Field	Description
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.

Item	Description
-1	Indicates that the buffer is empty or the user location is not valid for one of these three reasons: • The uio_segflg field indicates user space and the base address (iov_base field) points to a location outside of the user address space. • The user does not have sufficient authority to access the location. • An I/O error occurred while the location was being accessed.

Related reference:

"uiomove Kernel Service" on page 516

"uphysio Kernel Service" on page 523

"ureadc Kernel Service" on page 529

v

The following kernel services begin with the letter v.

validate_pag or validate_pag64 Kernel Service

Purpose

Validates the Process Authentication Group (PAG) value.

Syntax

```
#include <sys/cred.h>

int validate_pag ( type, pg, npags )
int type;
struct paglist pg[];
int npags;

int validate_pag64 ( type, pg, npags )
int type;
struct paglist64 pg[];
int npags;
```

Parameters

Item	Description
type	PAG type to validate
pg	PAG list (must be in pinned memory)
npags	Number of PAGs to validate

Description

The **validate_pag** or **validate_pag64** kernel service validates the PAGs specified in *pg*. These services support the garbage collection of data structures by kernel extensions associated with PAGs. These structures are associated with a **set_pag** interface process. PAG values are inherited from parent to child across the **fork** system call, so one kernel extension structure can map to many processes. This routine is required to synchronize the execution of forks so that the process table can be scanned to identify a particular PAG. The **validate_pag** and **validate_pag64** kernel services cannot be used simultaneously with the **set_pag** interface. The application is required to provide this synchronization.

The value of *type* must be a defined PAG ID. The PAG ID for the Distributed Computing Environment (DCE) is 0. The *pg* parameter must be a valid, referenced PAG list in pinned memory.

Execution Environment

The **validate_pag** and **validate_pag64** kernel services can be called from the process environment only.

Return Values

A value of `0` is returned upon successful completion. Upon failure, a `-1` is returned and **errno** is set to a value that explains the error.

Error Codes

The **validate_pag** and **validate_pag64** kernel services fail if the following condition is true:

Item	Description
EINVAL	Invalid PAG specification

Related Information

Security Kernel Services in *Kernel Extensions and Device Support Programming Concepts*.

Related information:

Security Kernel Services

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

Item	Description
levsublev	Represents the value returned by **vec_init** kernel service when the virtual interrupt handler was defined.

Description

The **vec_clear** kernel service is not part of the base kernel but is provided by the device queue management kernel extension. This queue management kernel extension must be loaded into the kernel before loading any kernel extensions referencing these services.

The **vec_clear** kernel service removes the association between a virtual interrupt handler and the virtual interrupt level and sublevel that was assigned by the **vec_init** kernel service. The virtual interrupt handler at the sublevel specified by the *levsublev* parameter no longer registers upon return from this routine.

Execution Environment

The **vec_clear** kernel service can be called from the process environment only.

Return Values

The **vec_clear** kernel service has no return values. If no virtual interrupt handler is registered at the specified sublevel, no operation is performed.

Related reference:
"vec_init Kernel Service"

vec_init Kernel Service

Purpose

Defines a virtual interrupt handler.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int vec_init ( level,  routine,  arg)
int level;
void (*routine) ();
int arg;
```

Parameters

Item	Description
level	Specifies the virtual interrupt level. This level value is not used by the **vec_init** kernel service and implies no relative priority. However, it is returned with the sublevel assigned for the registered virtual interrupt handler.
routine	Identifies the routine to call when a virtual interrupt occurs on a given interrupt sublevel.
arg	Specifies a value that is passed to the virtual interrupt handler.

Description

The **vec_init** kernel service is not part of the base kernel but provided by the device queue management kernel extension. This queue management kernel extension must be loaded into the kernel before loading any kernel extensions referencing these services.

The **vec_init** kernel service associates a virtual interrupt handler with a level and sublevel. This service searches the available sublevels to find the first unused one. The *routine* and *arg* parameters are used to initialize the open sublevel. The **vec_init** kernel service then returns the level and assigned sublevel.

There is a maximum number of available sublevels. If this number is exceeded, the **vec_init** service halts the system. This service should be called to initialize a virtual interrupt before any device queues using the virtual interrupt are created.

The *level* parameter is not used by the **vec_init** service. It is provided for compatibility reasons only. However, its value is passed back intact with the sublevel.

Execution Environment

The **vec_init** kernel service can be called from the process environment only.

Return Values

The **vec_init** kernel service returns a value that identifies the virtual interrupt level and assigned sublevel. The low-order 8 bits of this value specify the sublevel, and the high-order 8 bits specify the level. The **attchq** kernel service uses the same format. This level value is the same value as that supplied by the *level* parameter.

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

Item	Description
vfsp	Points to a virtual file system structure.

Description

The **vfsrele** kernel service releases all resources associated with a virtual file system.

When a file system is unmounted, the **VFS_UNMOUNTED** flag is set in the **vfs** structure, indicating that it is no longer valid to do path name-related operations within the file system. When this flag is set and a **vnop_rele** v-node operation releases the last active v-node within the file system, the **vnop_rele** v-node implementation must call the **vfsrele** kernel service to complete the deallocation of the **vfs** structure.

Execution Environment

The **vfsrele** kernel service can be called from the process environment only.

Return Values

The **vfsrele** kernel service always returns a value of 0.

Related information:

Virtual File System Overview

Virtual File System (VFS) Kernel Services

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

Item	Description
vmhandle	Specifies the handle for the virtual memory object to be mapped.
offset	Specifies the offset in the virtual memory object and region.

Description

The **vm_att** kernel service performs the following tasks:

- Selects an unallocated region in the current address space and allocates it.
- Maps the virtual memory object specified by the *vmhandle* parameter with the access permission specified in the handle.
- Constructs the address in the current address space corresponding to the offset in the virtual memory object and region.

The **vm_att** kernel service assumes an address space model of fixed-size virtual memory objects and address space regions.

> **Attention:** If there are no more free regions, this call cannot complete and calls the **panic** kernel service.

Execution Environment

The **vm_att** kernel service can be called from either the process or interrupt environment.

Return Values

The **vm_att** kernel service returns the address that corresponds to the *offset* parameter in the address space.

Related reference:

“vm_det Kernel Service” on page 536

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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

Item	Description
eaddr	Specifies the starting address of the specified range.
nbytes	Specifies the number of bytes in the address range. If this parameter is negative or 0, no lines are invalidated.

Description

The **vm_cflush** kernel service writes to memory all modified cache lines that intersect the address range (*eaddr*, *eaddr* + *nbytes* -1). The *eaddr* parameter can have any alignment in a page.

The **vm_cflush** kernel service can only be called with addresses in the system (kernel) address space.

Execution Environment

The **vm_cflush** kernel service can be called from both the interrupt and the process environment.

Return Values

The **vm_cflush** kernel service has no return values.

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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

Item	Description
eaddr	Specifies the effective address in the current address space. The region containing this address is to be unmapped and deallocated.

Description

The **vm_det** kernel service unmaps the region containing the *eaddr* parameter and deallocates the region, adding it to the free list for the current address space.

The **vm_det** kernel service assumes an address space model of fixed-size virtual memory objects and address space regions.

> **Attention:** If the region is not mapped, or a system region is referenced, the system will halt.

Execution Environment

The **vm_det** kernel service can be called from either the process or interrupt environment.

Related reference:

"vm_att Kernel Service" on page 535

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_flushp Kernel Service

Purpose

Flushes the specified range of pages.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

int vm_flushp (*sid*, *pfirst*, *npages***)**
vmid_t *sid***;**
vpn_t *pfirst***;**
vpn_t *npages***;**

Parameters

Item	Description
sid	Identifies the base segment.
pfirst	The first page number within the range.
npages	The number of pages to flush starting from the *pfirst* value. All pages must be in the same segment.

Description

The **vm_flushp** kernel service routine initiates page-out for the specified page range in the virtual memory object. I/O is initiated for the modified pages only. If page-out is initiated, or the pages are currently undergoing page I/O, then they are flagged to have their page frames released upon completion. If the pages are not modified, their page frames are immediately released.

The caller can wait for the completion of I/O initiated by this and prior calls by calling the **vms_iowait** kernel service.

Note: The **vm_flushp** subroutine is not supported for use on large pages.

Execution Environment

The **vm_flushp** kernel service can be called from the process environment only.

This is intended for files, and might not be called for working storage segments.

Return Values

Item	Description
0	Indicates the completion of the flush operation.
EINVAL	Indicates one of the following errors: • *pfirst* = 0 and *npages* = 0. • *pfirst* < 0. • *npages* < 0. • Page interval not in one segment. • Invalid *sid* parameter. • Invalid segment type.

Related reference:
"vm_writep Kernel Service" on page 578
"vm_invalidatep Kernel Service" on page 543
Related information:
Understanding Virtual Memory Manager Interfaces

vm_galloc Kernel Service

Purpose

Allocates a region of global memory in the 64-bit kernel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vm_galloc (int  type, vmsize_t  size, ulong * eaddr)
```

Description

The **vm_galloc** kernel service allocates memory from the kernel global memory pool on the 64-bit kernel. The allocation size is rounded up to the nearest 4K boundary. The default page protection key for global memory segments is 00 unless overridden with the **V_UREAD** flag.

The type field may have the following values, which may be combined:

Item	Description
V_WORKING	Required. Creates a working storage segment.
V_SYSTEM	The new allocation is a global system area that does not belong to any application. Storage reference errors to this area will result in system crashes.
V_UREAD	Overrides the default page protection of 00 and creates the new region with a default page protection of 01.
V_NOEXEC	Pages in the region will have no-execute protection by default. Only supported on POWER4 and later hardware.

The **vm_galloc** kernel service is intended for subsystems that have large data structures for which **xmalloc** is not the best choice for management. The kernel **xmalloc** heap itself does reside in global memory.

Parameters

Item	Description
type	Flags that may be specified to control the allocation.
size	Specifies the size, in bytes, of the desired allocation.
eaddr	Pointer to where **vm_galloc** will return the start address of the allocated storage.

Execution Environment

The **vm_galloc** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Successful completion. A new region was allocated, and its start address is returned at the address specified by the **eaddr** parameter.
EINVAL	Invalid size or type specified.
ENOSPC	Not enough space in the **galloc** heap to perform the allocation.
ENOMEM	Insufficient resources available to satisfy the request.

Related reference:

"vm_gfree Kernel Service"

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_gfree Kernel Service

Purpose

Frees a region of global memory in the kernel previously allocated with the **vm_galloc** kernel service.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vm_gfree (ulong  eaddr, vmsize_t  size)
```

Description

The **vm_gfree** kernel service frees up a global memory region previously allocated with the **vm_galloc** kernel service. The start address and size must exactly match what was previously allocated by the **vm_galloc** kernel service. It is not valid to free part of a previously allocated region in the **vm_galloc** area.

Any I/O to or from the region being freed up must be quiesced before calling the **vm_gfree** kernel service.

Parameters

Item	Description
eaddr	Start address of the region to free.
size	Size in bytes of the region to free.

Execution Environment

The **vm_gfree** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Successful completion. The region was freed.
EINVAL	Invalid size or start address specified. This could mean that the region is out of range of the **vm_galloc** heap, was not previously allocated with **vm_galloc**, or does not exactly match a previous allocation from **vm_galloc**.

Related reference:

"vm_galloc Kernel Service" on page 538

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_guatt Kernel Service

Purpose

Attaches an area of global kernel memory to the current process's address space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vm_guatt (kaddr, size, key, flags, uaddr)
void * kaddr;
vmsize_t size;
vmkey_t key;
long flags;
void ** uaddr;
```

Parameters

Item	Description
kaddr	Kernel address to be attached (returned from **vm_galloc** when the global memory was allocated).
size	Length of the region to be inserted into the process address space, in bytes.
key	Protection key that the process will use when accessing the attached region.
flags	Type of **vm_guatt** operation; must be set to **VU_ANYWHERE**.
uaddr	Pointer to user space address where the region was attached by vm_guatt. The location pointed to by *uaddr* (**uaddr*) must be null when the **vm_guatt** call is made.

Description

vm_guatt is a kernel service used to attach a region of global kernel memory that was allocated with **vm_galloc** to a process's address space. If the call is successful, the address in the process address space where the memory was attached is returned in the location pointed to by *uaddr*.

key can be set to **VM_PRIV** or **VM_UNPRIV**. If it is set to **VM_PRIV**, the process will be able to read and write the attached region. If it is set to **VM_UNPRIV**, the process will not be able to write the region and will only be able to read it if the **vm_galloc** of the region was done with the **V_UREAD** flag on.

vm_guatt attachments are not inherited across a process fork.

Execution Environment

The **vm_guatt** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates one of the following errors: • *flags* or *key* is not set to a valid value, *size* is 0, or the value pointed to by *uaddr* is non-NULL. • Region indicated by *kaddr* and *size* does not lie within a region previously allocated by **vm_galloc**.

Implementation Specifics

The **vm_guatt** kernel service is part of Base Operating System (BOS) Runtime.

Related reference:

"vm_galloc Kernel Service" on page 538

"vm_gudet Kernel Service"

Related information:

Memory Kernel Services

vm_gudet Kernel Service

Purpose

Removes a region attached with **vm_guatt** from the current process's address space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vm_gudet (kaddr, uaddr, size, flags)
```

```
void * kaddr;
void * uaddr;
vmsize_t size;
long flags;
```

Parameters

Item	Description
kaddr	Kernel address attached by **vm_guatt**.
uaddr	Location in the process address space where the kernel region was attached.
size	Length of the attached region, in bytes.
flags	Type of **vm_gudet** operation, must be **VU_ANYWHERE**.

Description

vm_gudet is a kernel service that detaches a region of global kernel memory that was attached by **vm_guatt**. This memory must still be allocated, detaching a region after it has been deallocated with **vm_gfree** is an error. If the detach is successful, the global kernel memory region at *kaddr* will no longer be addressable at *uaddr* by the calling process.

Execution Environment

The **vm_gudet** kernel service can be called from the process environment only.

Return Values

Item	Description
0	User address detached successfully.
EINVAL	Indicates one of the following errors: • Invalid flags. • Region indicated by *kaddr* and *size* does not lie within a region allocated by **vm_galloc**.

Implementation Specifics

The **vm_gudet** kernel service is part of Base Operating System (BOS) Runtime.

Related reference:

"vm_galloc Kernel Service" on page 538

"vm_gfree Kernel Service" on page 539

"vm_guatt Kernel Service" on page 540

Related information:

Memory Kernel Services

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

Item	Description
vmid	Specifies a virtual memory object identifier, as returned by the **vms_create** kernel service.
key	Specifies an access key. This parameter has a 1 value for limited access and a 0 value for unlimited access, respectively.

Description

The **vm_handle** kernel service constructs a virtual memory handle for use by the **vm_att** kernel service. The handle identifies the virtual memory object specified by the *vmid* parameter and contains the access key specified by the *key* parameter.

A virtual memory handle is used with the **vm_att** kernel service to map a virtual memory object into the current address space.

The **vm_handle** kernel service assumes an address space model of fixed-size virtual memory objects and address space regions.

Execution Environment

The **vm_handle** kernel service can be called from the process environment only.

Return Values

The **vm_handle** kernel service returns a virtual memory handle type.

Related reference:

"vm_att Kernel Service" on page 535

"vms_create Kernel Service" on page 571

Related information:

Understanding Virtual Memory Manager Interfaces

vm_invalidatep Kernel Service

Purpose

Releases page frames in the specified range for a non-journaled persistent segment or client segment.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

int vm_invalidatep (*sid*, *pfirst*, *npages***)**
vmid_t *sid***;**
vpn_t *pfirst***;**
ulong *npages***;**

Parameters

Item	Description
sid	Identifies the base segment.
pfirst	The first page number within the range.
npages	The number of pages to invalidate starting from the *pfirst* value. All pages must be in the same segment.

Description

The **vm_invalidatep** kernel service routine discards any page frames associated with the virtual memory object in the specified page range.

If a page within the specified range is found in page-in or page-out state, then the thread is synchronously put to sleep until the page I/O completes. When the I/O is complete, any memory-resident page frame is then freed.

Note: The **vm_invalidatep** subroutine is not supported for use on large pages.

Execution Environment

The **vm_invalidatep** kernel service can be called from the process environment only.

This is intended for files, and might not be called for working storage segments.

Return Values

Item	Description
0	Indicates the completion of the invalidating operations.
EINVAL	Indicates one of the following errors: • *pfirst* < 0. • *npages* < 0. • Page interval not in one segment. • Invalid *sid* parameter. • Invalid segment type.

Related reference:
"vm_writep Kernel Service" on page 578
"vms_iowait, vms_iowaitf Kernel Services" on page 573
Related information:
Understanding Virtual Memory Manager Interfaces

vm_ioaccessp Kernel Service

Purpose

Initiates asynchronous page-in or page-out for the range of pages specified.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

int vm_ioaccessp (*bsid*, *pfirst*, *npages*, *modifier***)**
vmid_t *bsid***;**
vpn_t *pfirst***;**
vpn_t *npages***;**
uint *modifier***;**

Parameters

Item	Description
bsid	Identifies the base segment.
pfirst	The first page number within the range.
npages	The number of pages to access starting from the *pfirst* value. All pages must be in the same segment.
modifier	Flags passed in by the user. These flags are detailed below.

Description

The **vm_ioaccessp** kernel service routine enables a client file system with a thread-level strategy routine to access the pages specified. This call is strictly advisory and might return without having done anything. If you want to actually move the data, call the **vm_uiomove** kernel service. If you want to pre-page the target, then call the **vm_readp** kernel service.

The flags passed in through the *modifier* parameter determine what type of action taken by the **vm_ioaccessp** kernel service. For details of each flag's purpose, see the table below.

The flags carry certain restrictions. You cannot request both a make and a flush operation. Also, if the **VM_IOACCESSP_WAITONLY** flag is declared then you must specify at least one type of wait operation. Finally, you cannot request a make or a flush operation if the **VM_IOACCESSP_WAITONLY** flag is declared.

Flags

Value	Name	Purpose
0x0001	**VM_IOACCESSP_MAKE**	Creates new pages in the page-in state in the specified range. Can only make up to 1MB of pages.
0x0002	**VM_IOACCESSP_FLUSH**	Flushes pages in the specified range.
0x0004	**VM_IOACCESSP_PGINWAIT**	If a page in the specified range is in page-in state, then block until page-in is complete.
0x0008	**VM_IOACCESSP_PGOUTWAIT**	If a page in the specified range is in page-out state, then block until page-out is complete.
0x0010	**VM_IOACCESSP_WAITONLY**	Returns once the specified wait is complete. The **VM_IOACCESSP_PGINWAIT** flag and the **VM_IOACCESSP_PGOUTWAIT** flag must also be specified.

Execution Environment

The **vm_ioaccessp** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates the completion of the I/O access operations.

Item	Description
EINVAL	Indicates one of the following errors: • *pfirst* = 0 and *npages* = 0. • *pfirst* < 0. • *npages* < 0. • Page interval not in one segment. • Invalid *sid* parameter. • Page make requests > 1 MB. • Not a client file system. • Unsupported flag used. • Both the **VM_IOACCESSP_MAKE** and the **VM_IOACCESSP_FLUSH** flags are set. • The **VM_IOACCESSP_WAITONLY** flag is set and the **VM_IOACCESSP_PGINWAIT** flag or the **VM_IOACCESSP_PGOUTWAIT** flag is not set. • The **VM_IOACCESSP_WAITONLY** flag and the **VM_IOACCESSP_MAKE** flag or the **VM_IOACCESSP_FLUSH** flag are set.

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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

Item	Description
vmid	Specifies the ID of the virtual memory object.
pno	Specifies the page number in the virtual memory object.

Description

The **vm_makep** kernel service makes the page specified by the *pno* parameter addressable in the virtual memory object without requiring a page-in operation. The **vm_makep** kernel service is restricted to client storage.

The page is not initialized to any particular value. It is assumed that the page is completely overwritten. If the page is already in memory, a value of 0, indicating a successful operation, is returned.

Execution Environment

The **vm_makep** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates a virtual memory object type or page number that is not valid.
EFBIG	Indicates that the page number exceeds the file-size limit.

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_mem_policy System Call

Purpose

Allows callers to get or set their applications' default memory placement policies.

Library

Standard C Library (**libc.a**)

Syntax

#include <sys/rset.h>

#include <sys/vminfo.h>

int vm_mem_policy(int cmd, int *early_lru, int *policies, int num_policies)

Description

The **vm_mem_policy** system call allows callers to get or set their applications' default memory placement policies for different types of memory.

Following are the different types of placement policies:

Item	Description
P_FIRST_TOUCH	Places the memory at the **MCM** where the application first referenced it. This is also achieved by setting the **MEMORY_AFFINITY** environment variable to **MCM** and benefit the applications with an identified home **MCM** to run on.
P_BALANCED	Uses the stripe memory in the application across all the system's **MCM**s. This benefits applications that do not identify a home **MCM** to run on, or on global memory objects that is accessed by many applications.
P_DEFAULT	Accepts the system's default policy for memory placement, which can be either the first touch or balanced policy, depending on the circumstances and the type of memory.

The **vm_mem_policy** system call allows the caller to get or set the **early_lru** flag, which triggers the system to look for stealable pages immediately after a **P_FIRST_TOUCH** driven scan for local memory (the memory on the same **MCM** the application is running on) does not find any available pages.

The parameters *policies*, and *num_policies* allow a caller to fine control over the default memory placement policies of different types of memory. The policy settings take effect on any new memory page the application creates after having called this function. The existing memory pages of the application retains their existing memory placement.

Parameters

Item **Description**

cmd
A command that is either **VM_SET_POLICY** or **VM_GET_POLICY**. The **VM_GET_POLICY** command copies the current policy setting into the buffers supplied by the caller, and does not change any of the process policies. The **VM_SET_POLICY** command reads input from the supplied buffers and changes the process policies accordingly.

early_lru
A pointer to an integer that indicates the state of the *early_lru* setting for first touch policy. Enabling *early_lru* causes memory to be paged out in order to fulfill a first-touch request for memory placement.

The possible values for *early_lru* are:

0 turn off *early_lru*.

1 turn on *early_lru*.

-1 do not modify *early_lru* setting for **VM_SET_POLICY**.

policies
A pointer to an array of policies for distinct types of memory. Each array element contains one of the policy types. The array element contains -1 to leave the policy unchanged for the corresponding memory type. The array must be declared with a length of **VM_NUM_POLICIES**. The list that follows enumerates the memory types whose policies can be changed in the form of constants. Enter the constant that is an array index into the policies array for the corresponding memory type.

VM_POLICY_TEXT
policy for executable program text

VM_POLICY_STACK
policy for program stack

VM_POLICY_DATA
policy for program heap and private **mmap** data

VM_POLICY_SHM_NAMED
policy for shared memory obtained via **shm_open()** or **shmget()** with a key

VM_POLICY_SHM_ANON
policy for anonymous mmap memory, or shared memory obtained via **shmget()** with **IPC_PRIVATE** key

VM_POLICY_MAPPED_FILE
policy for files mapped into the address space via **shmat()** or **mmap()**

VM_POLICY_UNMAPPED_FILE
policy for open files that are not mapped

num_policies
Number of elements in the policies array. This value must be set to **VM_NUM_POLICIES**.

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

Item	Description
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

Item	Description
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.

Related reference:

"vm_umount Kernel Service" on page 576

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_mounte Kernel Service

Purpose

Adds a file system with a thread-level strategy routine to the paging device table.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

int vm_mounte (*in_dtype*, *in_devid*, *in_thrinfop***)**
int *in_dtype***;**
dev_t *in_devid***;**
struct thrpginfo * *in_thrinfop***;**

Parameters

Item	Description
in_dtype	Specifies the type of device. Supported device types are **D_REMOTE**, **D_LOGDEV**, **D_SERVER**, **D_LOCALCLIENT**. Other optional flags are detailed below.
in_devid	If the type is **D_LOGDEV**, specifies a dev_t object of the block device. If the type is **D_REMOTE** or **D_SERVER**, specifies a pointer to a strategy routine.
in_thrinfop	Pointer to a **thrpginfo** structure.

Description

The **vm_mounte** kernel service allocates an entry in the paging device table for the device specified. The **vm_mounte** kernel service can also mount a client file system with a thread-level strategy routine. This is done by passing in the **D_THRPGIO** and the **D_ENHANCEDIO** flags.

Flags

Name	Purpose
D_ENHANCEDIO	Indicates an enhanced I/O-aware file system.
D_PREXLATE	Enables pre-translation as the default for all but remote file systems.
D_THRPGIO	Indicates a thread-level strategy routine.

Execution Environment

The **vm_mounte** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
ENOMEM	Indicates that there is no memory for the **buf** or the **thrpginfo** structure.
EINVAL	Indicates one of the following errors: • The file system strategy pointer is already in the paging device table, or in case of **D_SERVER**, a server is already defined. • The *in_dtype* parameter is set to the **D_PAGING** or the **D_FILESYSTEM** value. • The **thrpginfo** structure has not been initialized correctly. • The **D_THRPGIO** flag has been set without the **D_ENHANCEDIO** flag.

Related reference:

"vm_umount Kernel Service" on page 576

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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

Item	Description
vmid	Specifies the virtual memory object ID.
offset	Specifies the offset in the virtual memory object.
limit	Indicates the limit on the transfer length. If this parameter is negative or 0, no bytes are transferred.
rw	Specifies a read/write flag that gives the direction of the move. The possible values for this parameter (**UIO_READ**, **UIO_WRITE**) are defined in the **/usr/include/sys/uio.h** file.
uio	Points to the **uio** structure.

Description

The **vm_move** kernel service moves data between a virtual memory object and the buffer specified in a **uio** structure.

This service determines the virtual addressing required for the data movement according to the offset in the object.

The **vm_move** kernel service is similar to the **uiomove** kernel service, but the address for the trusted buffer is specified by the *vmid* and *offset* parameters instead of as a **caddr_t** address. The offset size is also limited to the size of a **caddr_t** address since virtual memory objects must be smaller than this size.

Note: The **vm_move** kernel service does not support use of cross-memory descriptors.

I/O errors for paging space and a lack of paging space are reported as signals.

Execution Environment

The **vm_move** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EFAULT	Indicates a bad address.
ENOMEM	Indicates insufficient memory.
ENOSPC	Indicates insufficient disk space.
EIO	Indicates an I/O error.

Other file system-specific **errno** global variables are returned by the virtual file system involved in the move function.

Related reference:

"uiomove Kernel Service" on page 516

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_mvc Kernel Service

Purpose

Reads or writes partial pages of files.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

int vm_mvc (*in_sid*, *in_pno*, *in_pgoffs*, *in_count*, *in_cmd*, *in_xmemdp*, *in_ptr***)**
vmid_t *in_sid***;**
vpn_t *in_pno***;**
int *in_pgoffs***;**
int *in_count***;**
int *in_cmd***;**
struct xmem * *in_xmemdp***;**
void * *in_ptr***;**

Parameters

Item	Description
in_sid	The primary memory object, m1.
in_pno	The m1 pno object. If it is a read operation, this parameter refers to the source. If not, it refers to a target.
in_pgoffs	The byte offset in the pno object.
in_count	The number of bytes to zero or copy in memory.
in_cmd	The reason for the function call. The possible values could be Zero, Zero(protect), read, or write.
in_xmemdp	The xmem descriptor for the second memory object, m2.
in_ptr	The byte offset in the xmem object.

Description

The **vm_mvc** kernel service is meant to be used by client file systems doing read or write operations to partial pages of files, where the file is denoted by the m1 object and the read or write buffer by the m2 object. Such cases arise on EOF handling, fragments, compression, and holes among other situations.

Given two memory object, m1 and m2, the **vm_mvc** kernel service allows you to do one of the following operations:

- Zero out bytes on the m1 object (**VM_MVC_ZERO**).
- Zero out and protect the m1 object (**VM_MVC_PROTZERO**).
- Copy bytes from the m1 object to the m2 object (**VM_MVC_READ**).
- Copy bytes from the m2 object to the m1 object (**VM_MVC_WRITE**).

The first memory object, m1, is characterized by a *sid* parameter and a *pno* parameter. The second memory object, m2, is characterized by an xmem descriptor and a pointer for an offset. The second memory object is a user or kernel buffer.

Note: The second memory object must be pinned.

Flags

in_cmd	Purpose
VM_MVC_ZERO	Zeros out the bytes on the m1 object.
VM_MVC_READ	Copies bytes from the m1 object to the m2 object.
VM_MVC_WRITE	Copies bytes from the m2 object to the m1 object.
VM_MVC_PROTZERO	Zeros out and protects the m1 object.

Execution Environment

The **vm_mvc** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates that the I/O access operations completed successfully.
ENOENT	Indicates that the (sid, pno) set was not mapped to a real frame.
EINVAL	Indicates one of the following errors: • The m1 object crosses page boundary. • The *in_cmd* parameter does not contain a valid command.

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_pattr System Call and kvm_pattr Kernel Service

Purpose

Queries or modifies virtual memory attributes.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/vmpattr.h>

int vm_pattr (
long cmd,
pid_t pid,
void * attr,
size_t attr_size );

int kvm_pattr (
long cmd,
pid_t pid,
void * attr,
size_t attr_size );
```

Description

The **vm_pattr** system call queries or modifies memory attributes of the calling process's address space or that of another user process.

The **kvm_pattr** kernel service provides the same function to kernel subsystems (kernel extensions, kernel processes and so on) except that it cannot modify another kernel process' memory attributes.

Parameters

Item **Description**

cmd
The following commands can be passed in:

VM_PA_SET_PSIZE or VM_PA_GET_PSIZE
These commands set or retrieve the page size used for a specified memory range.

VM_PA_GET_RMUSAGE
This command retrieves the amount of the real memory in bytes being used for a specified memory range.

VM_PA_SET_PSPA or VM_PA_GET_PSPA
These commands set or retrieve the page size promotion aggressiveness factor for a specified memory range.

VM_PA_GET_PSPA_ALIGN
This command retrieves the minimum memory alignment necessary for memory ranges specified to the **vm_pattr** kernel service when using the **VM_PA_SET_PSPA** command.

VM_PA_CHECK_PSIZE
This command reports if a specified page size can be used for a memory range.

VM_PA_SET_LSA_POLICY
This command allows the shared memory address space allocator to be tuned according to a process requirements. This command should be run before any shared memory regions are created.

VM_PA_SET_PSIZE_EXTENDED or VM_PA_GET_PSIZE_EXTENDED
These commands provide variable large page size segment support.

pid
Specifies the ID of the process whose memory attributes are to be queried or modified. A value of -1 specifies the calling process. The root user can specify any process ID, but other users can only specify processes they own (that is, the target process's user ID must match the calling process's user ID).

The **vm_pattr** system call is only supported on user processes while the **kvm_pattr** kernel service can target user processes or its own kernel process (for example, *pid* = -1).

attr
A pointer to a structure describing the effective address range for the memory being queried or modified and additional data depending on the command.

The range is specified through the following **vm_pa_range** structure:

```
struct vm_pa_range
             {
                 ptr64_t  rng_start;
                 size64_t rng_size;
             };
```

The range specified must be in the target process's address space and must correspond to one of these process areas:

- Main program data (initialized, bss, or heap).
- Shared library data or private module load area data.
- Privately loaded text.
- Initial thread stack area.
- Anonymous shared memory (System V shared memory, extended System V shared through EXTSHM, and POSIX real-time shared memory). The target process must have write access to the memory in order to change the attributes of the shared memory range.
- Anonymous mmap memory.

If the memory range specified includes shared memory or mmap memory, the calling process must have write access to the memory according to the shared memory descriptor or mapping attributes in order to change the attributes of the range. The range can have additional restrictions based on the following commands.

Item **Description**

attr (continued) The structure specified through the *attr* parameter must be a pointer to one of the following structures:

VM_PA_SET_PSIZE or VM_PA_GET_PSIZE

These commands take a pointer to the following structure:

```
struct vm_pa_psize
              {
                     struct vm_pa_range pa_range;
                     psize_t            pa_psize;
              };
```

For the **VM_PA_SET_PSIZE** command, the *pa_psize* parameter is the page size (in bytes) to use for the given range. This is an advisory setting that might or might not be used at the operating system's discretion. This must be a valid page size between the minimum and maximum page sizes of all segments in the range. Additionally, the range must start and end on a multiple of the specified page size. If an error occurs during the processing of this command, any successfully altered page size settings can remain set.

For the **VM_PA_GET_PSIZE** command, the page size (in bytes) backing the specified memory range is returned in the *pa_psize* parameter. The range must start and end on a multiple of the smallest page size supported as reported by the **sysconf(_SC_PAGE_SIZE)** subroutine. If the range is using multiple page sizes, the smallest page size in the range is reported. Unlike the **VM_PAGE_INFO** command of the **vmgetinfo** subroutine that reports the segment's base page size, the page size reported by the **VM_PA_GET_PSIZE** command is the actual page size being used at the time the **vm_pattr** system call was called. The page size reported is transient because the operating system can change the backing page size at any time. Therefore, the page size reported must be for informational purposes only.

VM_PA_SET_PSIZE_EXTENDED

This command takes a pointer to the following structure:

```
struct vm_pa_psize_extended
    {
     struct vm_pa_range   pa_range;
     psize_t              pa_psize;
     size_t               pa_info_size;
     uint64_t            *pa_info;
    }
```

This command is essentially the same as **VM_PA_SET_PSIZE** except that *pa_psize* must be 16 MB and, if not NULL, *pa_info* can be used to pass additional information specifying one or more affinity domains.

The info passed by the parameter is advisory request, and the system might choose to ignore it.

The *pa_range* is scanned for subregions that begin and end on a 16 MB boundary, are fully backed with 4 KB or 64 KB pages, and have uniform page attributes. The page attributes include read or write page protection, storage key protection, and no-execute protection.

The data in qualifying 16 MB subregions is colocated to a 16 MB contiguous block of physical memory, and it uses 16 MB hardware translations.

If the *pa_info* pointer is NULL, the memory for collocation is allocated from any memory SRAD, affinity domain chosen by the operating system.

If parameter value is not NULL, *pa_info* must point to an *rsethandle_t* that describes a set of affinity domains from which the physical memory for the collocation must be allocated. The object should be allocated by a call to *rs_alloc* (RS_EMPTY). It must then be initialized with one *rs_op*(RS_ADDRESOURCE, ..., R_MEMPS, srad#) call per affinity domain being requested.

This command can potentially affect system performance and is not generally recommended; therefore, this command requires you to have either the CAP_BYPASS_RAC_VMM and CAP_PROPAGATE capabilities or root authority.

VM_PA_GET_PSIZE_EXTENDED

This command is essentially the same as the **VM_PA_GET_PSIZE** command except that it can also return the 64 KB and 16 MB subregions that are using an hardware translation page size different from the underlying segments default page size.

Item **Description**

attr (continued)

If the *pa_info* field is NULL, this command is identical to the **VM_PA_GET_PSIZE** command.

The *pa_info* field should point to an array containing two 64 bit integers. The *pa_info_size* field should be set to the size of the array.

In the first 64-bit integer, this command reports the number of 64-KB sized and aligned subregions in the specified *pa_range* range that consist of 16 contiguous 4-KB pages that are promoted to using a 64-KB page size hardware translation. In the second 64-bit integer, this command reports the number of 16 MB sized and aligned subregions in the specified range that consist of either 4096 4 KB or 256 64-KB contiguous pages that are promoted to using a 16-MB page size hardware translation.

The *pa_psize* field reports the smallest page size found for the specified range.

The information reported is transient because the operating system can change the backing page size at any time. Therefore, the page size reported must be for informational purposes only.

attr (continued)

VM_PA_GET_RMUSAGE

This command takes a pointer to the following structure:

```
struct vm_pa_rmusage
          {
                struct vm_pa_range pa_range;
                size64_t           pa_rbytes;
          };
```

This command reports the amount of real memory (in bytes) used for the given range in the *pa_rbytes* field. This can help an application decide whether it needs to use a large page size for a specific range based on how much real memory the range is using. For example, if a 64KB range is only using 4KB of real memory, then it does not make sense to try to use a 64KB page size for that range. But if it is using all 64KB or some large percentage of it, then the application might decide to use a 64KB page size. The range specified for this command has no alignment requirements for this command, and the command includes only those bytes in the range that are using real memory.

VM_PA_SET_PSPA or VM_PA_GET_PSPA

These commands take a pointer to the following structure:

```
struct vm_pa_pspa
          {
                struct vm_pa_range pa_range;
                int    pa_pspa;
          };
```

The **VM_PA_SET_PSPA** command can set the page size promotion aggressiveness for the specified range. The *pa_pspa* setting is in the same units as the vmm_default_pspa vmo tunable. This setting is the inverse of the real memory occupancy threshold needed to promote to a large page size and ranges from -1 to 100. The value of -1 indicates that no page promotion can occur regardless of the occupancy of the memory range. A value of 0 indicates a page size promotion can only be done when the memory range is fully occupied. A value of 100 indicates a page promotion must be done at the first reference to the memory range.

This setting is only supported at a segment granularity, so the range must start and end on a segment boundary. The alignment requirement for the range can be found using the **VM_PA_GET_PSPA_ALIGN** command with the **vm_pattr** system call.

If an error occurs during the processing of the **VM_PA_SET_PSPA** command, the **vm_pattr** system call can return after altering the page size promotion thresholds for part of the specified range.

The **VM_PA_GET_PSPA** command retrieves the page size promotion aggressiveness factor for the specified range. If the range spans multiple segments consisting of different page promotion thresholds, the *pa_pspa* field is updated with the least aggressive PSPA setting (the smallest PSPA setting across all of the segments).

The PSPA commands are not supported on mmap or EXTSHM memory ranges.

Item **Description**

attr (continued)

VM_PA_GET_PSPA_ALIGN

This command takes a pointer to the following structure:

```
 struct vm_pa_pspa_align
              {
                     struct vm_pa_range pa_range;
                     size64_t pa_pspa_align;
              };
```

The **VM_PA_GET_PSPA_ALIGN** command returns the minimum memory alignment requirements of a memory range for the **VM_PA_SET_PSPA** command in the *pa_pspa_align* field based on what segments are contained in the specified memory range. If a memory range spans segments with different alignment requirements, this command returns the largest of the alignment requirements.

The alignment requirements for the **VM_PA_SET_PSPA** command are as follows:

attr (continued)

VM_PA_SET_LSA_POLICY

This command takes a pointer to the following structure:

```
struct vm_pa_lsa_options
 {
 u_int64_t setting;
 size64_t value;
 };
```

The following settings are allowed:

VM_PA_SHM_1TB_SHARED

This setting controls the threshold of the number of 256 MB segments required before a SHM object is considered big enough to be placed in its own 1 TB region to be promoted to the large alias segments. Values can range from 0 to 4 KB.

VM_PA_SHM_1TB_UNSHARED

This setting controls the threshold of the number of 256 MB segments required before a group of SHM object packed in a 1 TB aligned group is promoted to the large alias segments. Values can range from 0 to 4 KB.

attr (continued)

Process's Memory Area Minimum Alignment

Main process data 256 MB

Process stack 256 MB

Shared Library data 256 MB

Privately loaded module data 256 MB

Privately loaded module text 256 MB

POSIX Real-Time Shared Memory 256 MB

Anonymous MMAP 256 MB

Anonymous Extended System V Shared memory 256 MB

Anonymous System V Shared memory with page sizes less than or equal to 256 MB 256 MB

Anonymous System V shared memory backed with 16 GB page size 1 TB

Item **Description**

attr (continued)

VM_PA_CHECK_PSIZE

This command takes a pointer to the following structure:

```
struct vm_pa_psize_check
          {
               struct vm_pa_range pa_range;
               psize_t            pa_psize;
               int                pa_reason;
          };
```

The **VM_PA_CHECK_PSIZE** command determines if a specific page size is allowed by the **VM_PA_SET_PSIZE** command for a specified memory range. The **VM_PA_CHECK_PSIZE** command can be used when the application wants more detailed information about why a **VM_PA_SET_PSIZE** operation fails, or to check if a **VM_PA_SET_PSIZE** operation will successfully modify the page size for the range specified.

This command must be used on a memory range that spans a single page and is aligned to the page size specified by the *pa_psize* parameter. If the page size can be used for that range, the *pa_reason* parameter is set to 0. Otherwise, it is set to a reason code defined in the **vmpattr.h** header file.

VMPATTR_SET_PSIZE_VALID The specified page size can be used for the specified range.

VMPATTR_INVALID_MPSS_PSIZE The specified page size is not supported in mixed page size segments.

VMPATTR_NON_MPSS_SEGMENT The address range specified is from a segment that does not support mixed page sizes.

VMPATTR_NON_MPSS_PAGE The size of the target page cannot be modified. For example, this reason code can be returned when trying to set an address range to a 64 KB page size if a portion of the range has page protection settings that do not match the rest of the range.

VMPATTR_RDONLY_MEM The target range cannot be modified because the caller does not have write access to the memory specified.

VMPATTR_PAGE_ATTRIBUTES The address range specified does not have uniform page attributes.

VMPATTR_NOT_FULLY_POPULATED The address range specified does not fully reside in memory.

VMPATTR_PHYSICAL_ATTACHMENTS The address range specified has memory affinity attachments that specify more than one affinity domain.

VMPATTR_MEMORY_TYPE_UNSUPPORTED The address range contains a memory object that does not support the requested page size in a mixed page size segment.

attr_size The *attr_size* parameter must be the size of the structure needed, or greater for the specified command.

Return Values

When successful, these commands return 0. Otherwise, they return -1 and set the errno global variable to indicate the error.

Error Codes

Item **Description**

EPERM
The calling process does not have the appropriate privilege to perform the requested operation.

ESRCH
The target process does not exist or is not in a valid state.

ENOMEM
The range specified contains a hole. A hole is any part of the target process's address space that is not backed by a virtual memory segment or is outside of the valid range of the virtual memory segment specified.

ENOTSUP
Any of the following situations can cause the **ENOTSUP** error:

- The target process is a kernel process other than the calling process.
- The command specified was the **VM_PA_SET_PSIZE** command and the page size specified is not supported for multiple page size segments.
- The command specified was either the **VM_PA_GET_PSPA** or the **VM_PA_SET_PSPA** command and the specified memory range includes mmap or EXTSHM segment(s).

EINVAL
Any of the following situations can cause the **EINVAL** error:

- The *attr_size* parameter specified is less than the size of the structure needed for this command.
- The range specified is outside the process's address space (for example, global kernel memory).
- The command specified was the **VM_PA_SET_PSIZE** command and the page size specified was not a valid page size supported by the system.
- The command specified was the **VM_PA_SET_PSPA** command and the address range specified was not aligned to the segment size backing the range.
- The command specified was the **VM_PA_SET_PSPA** command and the page promotion aggressiveness factor specified was not valid.
- The command specified was the **VM_PA_CHECK_PSIZE** command and the address range specified was not aligned to the page size specified.

ENOMEM
The command specified was **VM_PA_SET_PSIZE_EXTENDED**, and the system was unable to allocate memory from the set of affinity domains specified by the *pa_info* object or the entire set of system affinity domains without potentially causing a performance degradation.

EFAULT
The command specified was either **VM_PA_SET_PSIZE_EXTENDED**, or **VM_PA_GET_PSIZE_EXTENDED** and the *pa_info* address is not valid and is not NULL.

EINVAL
The command specified was either **VM_PA_SET_PSIZE_EXTENDED**, or **VM_PA_GET_PSIZE_EXTENDED** and the *pa_info* field is not-NULL, but the *pa_info_size* field is 0.

ENODEV
The command specified was **VM_PA_SET_PSIZE_EXTENDED**, and an invalid sradid was specified in *pa_info*.

Related information:

Dynamic variable page size support

vm_protect_kkey Kernel Service

Purpose

Sets kernel-key on a kernel address range.

Syntax

```
#include <sys/types.h>
#include <sys/skeys.h>
#include <sys/vmuser.h>

kerrno_t vm_protect_kkey (eaddr, nbytes, kkey, flags)
void * eaddr;
size_t nbytes;
kkey_t kkey;
unsigned long flags;
```

Parameters

Item	Description
eaddr	Starting address to protect.
nbytes	Number of bytes to protect.
kkey	Kernel-key value to set on memory.
flags	Defined flag value is: • **VMPK_NO_CHECK_AUTHORITY** – This flag indicates that extended authority checking will not be performed.

Description

The **vm_protect_kkey()** kernel service is used to alter the kernel-key associated with a virtual memory range. If set, any code that references the memory needs to include the kernel-key in their active keyset. The kernel-key is set for all pages in the effective address range specified by *eaddr* to *eaddr* + *nbytes* - *1*. If the address range does not specify a page-aligned area consisting of an integral number of full pages, an error will be returned.

By default, an authority check is performed when altering storage-keys. This check requires that the **vm_protect_kkey()** caller has write access to the pages' current kernel-key(s). This authority checking can be overridden by setting the **VMPK_NO_CHECK_AUTHORITY** value, but this is not recommended since the check can protect against some programming errors.

Execution Environment

The **vm_protect_kkey** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Successful.
EINVAL_VM_PROTECT_KKEY	Invalid parameter or execution environment.
EINVAL_VM_PROTECT_KKEY_PPAGE	Request includes a partial page.
EFAULT_VM_PROTECT_KKEY	Invalid address range.
EPERM_VM_PROTECT_KKEY	Insufficient authority to perform the operation.

If the **vm_protect_kkey()** kernel service is unsuccessful because of a condition other than that specified by the **EINVAL_VM_PROTECT_KKEY** error code, the kernel-key for some pages in the (*eaddr*, *eaddr* + *nbytes* - *1*) range might have been changed.

Related reference:

"vm_setseg_kkey Kernel Service" on page 569

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;
```

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.

Note: The **vm_protectp** subroutine is not supported for use on large pages.

Parameters

Item	Description
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.

Execution Environment

The **vm_protectp** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates one of the following errors: • Invalid virtual memory object ID. • The starting page in the designated page range is negative. • The number of pages in the page range is negative. • The designated page range exceeds the size of virtual memory object. • The target page range does not exist. • One or more large pages lie in the target page range.

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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

Item	Description
vmid	Specifies the ID of the virtual memory object to check.

Description

The **vm_qmodify** kernel service performs two tests to determine if a mapped file has been changed:

- The **vm_qmodify** kernel service first checks the virtual memory object modified bit, which is set whenever a page is written out.
- If the modified bit is 0, the list of page frames holding pages for this virtual memory object are examined to see if any page frame has been modified.

If both tests are false, the **vm_qmodify** kernel service returns a value of False. Otherwise, this service returns a value of True.

If the virtual memory object modified bit was set, it is reset to 0. The page frame modified bits are not changed.

Execution Environment

The **vm_qmodify** kernel service can be called from the process environment only.

Return Values

Item	Description
FALSE	Indicates that the virtual memory object has not been modified.
TRUE	Indicates that the virtual memory object has been modified.

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_qpages Kernel Service

Purpose

Returns the number of in-memory page frames associated with the virtual memory object.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

vpn_t vm_qpages (*sid***)**
vmid_t *sid***;**

Parameters

Item	Description
sid	Identifies the base segment.

Description

The **vm_qpages** kernel service routine returns the number of page frames associated with the virtual memory object with the *sid* parameter specified.

Execution Environment

The **vm_qpages** kernel service can be called from the process environment only.

This function can be run for persistent, client, and working storage segments.

Return Values

Item	Description
npages	The number of page frames.
-1	Indicates an invalid *sid* parameter.

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_readp Kernel Service

Purpose

Initiates asynchronous page-in for the range of pages specified.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

int vm_readp (*sid*, *pfirst*, *npages*, *flags***)**
vmid_t *sid***;**
vpn_t *pfirst***;**
vpn_t *npages***;**
int *flags***;**

Parameters

Item	Description
sid	Identifies the base segment.
pfirst	The first page number within the range.
npages	The number of pages to read starting from the *pfirst* value. All pages must be in the same segment, unless the `V_READMAKE` flag is used.
flags	Flags used by the function.

Description

The **vm_readp** kernel service routine starts the process of paging within the range of specified pages. This call is strictly advisory and might return without performing any operations.

The *flags* parameter is optional and accepts the following values:

V_IOWAIT
Instructs the **vm_readp** kernel service to wait for any page I/O requests to complete, within the range of specified pages, before initiating the read operation.

V_READMAKE
Instructs the **vm_readp** kernel service to create the segments within the range of the **vm_readp** operation.

Execution Environment

The **vm_readp** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates that the I/O access operations completed successfully.
EINVAL	Indicates one of the following errors: • *pfirst* = 0 and *npages* = 0. • *pfirst* < 0. • *npages* < 0. • Page interval > Maximum file size. • The *sid* parameter is not valid. • Not a file or persistent storage segment.

Related reference:
"vm_writep Kernel Service" on page 578
Related information:
Memory Kernel Services
Understanding Virtual Memory Manager Interfaces

vm_release Kernel Service

Note: The **vm_release** subroutine is not supported for use on large pages.

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;
```

Description

The **vm_release** kernel service releases pages that intersect the specified address range from the *vaddr* parameter to the *vaddr* parameter plus the number of bytes specified by the *nbytes* parameter. The value in the *nbytes* parameter must be nonnegative and the caller must have write access to the pages specified by the address range.

Each page that intersects the byte range is logically reset to 0, and any page frame is discarded. A page frame in I/O state is marked for discard at I/O completion. That is, the page frame is placed on the free list when the I/O operation completes.

Note: All of the pages to be released must be in the same virtual memory object.

Parameters

Item	**Description**
vaddr	Specifies the address of the first byte in the address range to be released.
nbytes	Specifies the number of bytes to be released.

Execution Environment

The **vm_release** kernel service can be called from the process environment only.

Return Values

Item **Description**

0 Indicates successful completion.

EACCES Indicates that the caller does not have write access to the specified pages.

EINVAL Indicates one of the following errors:

- The specified region is not mapped.
- The specified region is an I/O region.
- The length specified in the *nbytes* parameter is negative.
- The specified address range crosses a virtual memory object boundary.
- One or more large pages lie in the target page range.

Related reference:

"vm_releasep Kernel Service"

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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;
```

Description

The **vm_releasep** kernel service releases pages for the specified page range in the virtual memory object. The values in the *pfirst* and *npages* parameters must be nonnegative.

Each page of the virtual memory object that intersects the page range (*pfirst*, *pfirst* + *npages* -1) is logically reset to 0, and any page frame is discarded. A page frame in the I/O state is marked for discard at I/O completion.

For working storage, paging-space disk blocks are freed and the storage-protect key is reset to the default value.

Note: All of the pages to be released must be in the same virtual memory object.

Note: The **vm_releasep** subroutine is not supported for use on large pages.

Parameters

Item	Description
vmid	Specifies the virtual memory object identifier.
pfirst	Specifies the first page number in the specified page range.
npages	Specifies the number of pages in the specified page range.

Execution Environment

The **vm_releasep** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates one of the following errors: • An invalid virtual memory object ID. • The starting page is negative. • Number of pages is negative. • Page range crosses a virtual memory object boundary. • One or more large pages lie in the target page range.

Related reference:

"vm_release Kernel Service" on page 564

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_segmap Kernel Service

Purpose

Creates the segments associated with a range of bytes in a file and attaches them to the kernel's address space.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

int vm_segmap (*basesid*, *pfirst*, *flags*, *basepp***)**
vmid_t *basesid***;**
vpn_t *pfirst***;**
uint *flags***;**
caddr_t * *basepp***;**

Parameters

Item	Description
basesid	Identifies the base segment.
pfirst	The first page number within the range. The *pfirst* parameter is non-negative.
flags	Optional flags passed in by the user. .
basepp	The offset of the object to be attached.

Description

The **vm_segmap** kernel service routine creates segments associated with a range of bytes in a file. Afterwards, it uses the **vm_att** kernel service to map the specified virtual memory object to a region in the virtual address space and returns the effective address of that object in the *basepp* parameter.

Execution Environment

The **vm_segmap** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
caddr_t	The effective address of the attached object.
EINVAL	Indicates one of the following errors: • *pfirst* < 0. • Invalid *sid* parameter.
EFBIG	Indicates the range of values is too large to create.

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_setdevid Kernel Service

Purpose

Modifies the paging device table entry for a virtual memory object.

Syntax

```
#include <sys/types.h>
#include <sys/kerrno.h>
#include  <sys/vmuser.h>

kerrno_t vm_setdevid ( vmid, type,  ptr,  flags)
vmid_t vmid;
int type;
int (*ptr)();
unsigned long flags;
```

Parameters

Item	Description
vmid	Specifies the identifier for the virtual memory object for which the paging device table entry is to be set.
type	Specifies the type of device. The *type* parameter must have a value of D_REMOTE.
ptr	Points to the strategy routine of the file system.
flags	Reserved. You must set the *flags* parameter to zero.

Description

The **vm_setdevid** kernel service binds the paging device table entry associated with the file system strategy routine *ptr*, to the virtual memory object *vmid*. The paging device table entry must have already been mounted as type D_REMOTE through a prior **vm_mount** kernel service call.

After the file system has called the **vm_setdevid** kernel service on a given virtual memory object, subsequent paging I/O will be performed to or from the newly specified paging device table. Any outstanding I/O's to the paging device table formerly associated with the virtual memory object, remain queued, and will complete asynchronously. After they complete, subsequent paging I/O to those file pages will be performed to or from the newly specified paging device table.

The paging device table entry currently associated with the *vmid* object, on input to this call, must be valid and of type D_REMOTE. Any flags specified when the **vm_mount** kernel service gets called must match exactly any flags specified when the **vm_mount** kernel service gets called for the new paging device table entry.

Execution Environment

The **vm_setdevid** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL_VM_SETDEVID1	Indicates that the *vmid* value is not a client segment, or the input type does not have the value of D_REMOTE.
ENODEV_VM_SETDEVID2	Indicates that a file system with the strategy routine designated by the *ptr* parameter is not in the paging device table.
EINVAL_VM_SETDEVID3	Indicates that the new paging device table entry is not D_REMOTE or is not valid.
EINVAL_VM_SETDEVID4	Indicates that the paging device table entry currently associated with the *vmid* object is not D_REMOTE or is not valid.
EINVAL_VM_SETDEVID5	Indicates that the **vm_mount** flags for the current and new paging device table entries differ.
EINVAL_VM_SETDEVID6	Indicates that this was called at interrupt level.
EINVAL_VM_SETDEVID7	Indicates that the input flags was nonzero.
EINVAL_VM_SETDEVID8	Indicates that the input *vmid* value is not valid.

Related Information

The **vm_mount** kernel service, **vm_umount** kernel service.

Memory Kernel Services and Understanding Virtual Memory Manager Interfaces in *Kernel Extensions and Device Support Programming Concepts*.

Related reference:

"vm_mount Kernel Service" on page 548

"vm_umount Kernel Service" on page 576

Related information:

Understanding Virtual Memory Manager Interfaces

vm_setseg_kkey Kernel Service

Purpose

Sets the default kernel-key for a segment.

Syntax

```
#include <sys/types.h>
#include <sys/kerrno.h>
#include <sys/vmuser.h>

kerrno_t vm_setseg_kkey (vmid, kkey)
vmid_t vmid;
kkey_t kkey;
```

Parameters

Item	Description
vmid	Virtual memory object to act on.
kkey	New kernel key for the virtual memory object.

Description

The **vm_setseg_kkey** kernel service alters the default kernel-key for newly allocated pages in a segment. The kernel-key values for any existing pages in the segment are left unchanged.

Execution Environment

The **vm_setseg_kkey** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Successful.
EINVAL_VM_SETSEG_KKEY	Invalid parameter or execution environment.

Related reference:

"vm_protect_kkey Kernel Service" on page 559

vm_thrpgio_pop Kernel Service

Purpose

Retrieves the latest context information.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

void vm_thrpgio_pop (*in_ctxp***)**
ut_pgio_context_t * *in_ctxp***;**

Parameters

Item	Description
in_ctxp	The context structure used by the function.

Description

The **vm_thrpgio_pop** kernel service enables a client file system with a thread-level strategy routine to copy information from a context structure to the current thread. Afterwards, it makes the current thread point to the next context.

This service must be called if a client file system using a thread-level strategy routine has re-entered the Virtual Memory Manager and wishes to return to its strategy routine. This service restores the context that was saved using the **vm_thrpgio_push** kernel service.

Execution Environment

The **vm_thrpgio_pop** kernel service can only be used by client file systems using a thread-level strategy routine.

Return Values

The **vm_thrpgio_pop** kernel service has no return values.

Related reference:

"vm_thrpgio_push Kernel Service"

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_thrpgio_push Kernel Service

Purpose

Saves some context information of the current thread.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

void vm_thrpgio_push (*in_ctxp***)**
ut_pgio_context_t * *in_ctxp***;**

Parameters

Item	Description
in_ctxp	The context structure used by the function.

Description

The **vm_thrpgio_push** kernel service enables a client file system with a thread-level strategy routine to save information about the current thread to a linked list. The linked list is a Last-In-First-Out (LIFO) (stack) data structure, and is pointed to by the thread.

This service must be called if a client file system using a thread-level strategy routine has had its strategy routine invoked and wishes to re-enter the Virtual Memory Manager. This could involve a page fault on one of its client segments, or the use of one of the Virtual Memory Manager (VMM) services that operates on client segments.

The **vm_thrpgio_pop** kernel service must be invoked when all such Virtual Memory Manager callbacks are complete.

Execution Environment

The **vm_thrpgio_push** kernel service can only be used by client file systems using a thread-level strategy routine.

Return Values

The **vm_thrpgio_push** kernel service has no return values.

Related reference:

"vm_thrpgio_pop Kernel Service" on page 569

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vms_create Kernel Service

Purpose

Creates a virtual memory object of the specified type, size, and limits.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>

int vms_create (vmid, type, devgno, size, uplim, downlim)
vmid_t * vmid;
int   type;
dev_t devgno;
int   size;
int   uplim;
int   downlim;
```

Parameters

Item	Description
vmid	Points to the variable in which the virtual memory object identifier is to be stored.
type	Specifies the virtual memory object type and options as an OR of bits. The *type* parameter must have the value of **V_CLIENT**. The **V_INTRSEG** flag specifies if the process can be interrupted from a page wait on this object.
devgno	Specifies the address of the g-node for client storage. If the *type* parameter has the value of **V_CLIENT**, the third argument is a g-node *ptr* argument, otherwise, it is a *devgno* argument.
size	Specifies the current size of the file (in bytes). This can be any valid file size. If the V_LARGE is specified, it is interpreted as number of pages.
uplim	Ignored. The enforcement of file size limits is done by comparing with the **u_limit** value in the **u** block.
downlim	Ignored.

Description

The **vms_create** kernel service creates a virtual memory object. The resulting virtual memory object identifier is passed back by reference in the *vmid* parameter.

The *size* parameter is used to determine the size in units of bytes of the virtual memory object to be created. This parameter sets an internal variable that determines the virtual memory range to be processed when the virtual memory object is deleted.

An entry for the file system is required in the paging device table when the **vms_create** kernel service is called.

Execution Environment

The **vms_create** kernel service can be called from the process environment only.

Return Values

Item	Description
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.

Related reference:

"vms_delete Kernel Service"

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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

Item	Description
vmid	Specifies the ID of the virtual memory object to be deleted.

Description

The **vms_delete** kernel service deallocates the temporary resources held by the virtual memory object specified by the *vmid* parameter and then frees the control block. This delete operation can complete asynchronously, but the caller receives a synchronous return code indicating success or failure.

Releasing Resources

The completion of the delete operation can be delayed if paging I/O is still occurring for pages attached to the object. All page frames not in the I/O state are released.

If there are page frames in the I/O state, they are marked for discard at I/O completion and the virtual memory object is placed in the iodelete state. When an I/O completion occurs for the last page attached to a virtual memory object in the iodelete state, the virtual memory object is placed on the free list.

Execution Environment

The **vms_delete** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates that the *vmid* parameter is not valid.

Related reference:

"vms_create Kernel Service" on page 571

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vms_iowait, vms_iowaitf Kernel Services

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;
int vms_iowaitf ( vmid, flags)
vmid_t vmid;
int flags;
```

Parameter

Item	Description
vmid	Identifies the virtual memory object for which to wait.
flags	Optional flags passed in by the user.

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.

The **vms_iowait** function is a special case of the **vms_iowaitf** function with the **V_WAITERR** flag set.

Flags

Name	Purpose
V_WAITERR	Waits until the completion of all I/O unless an error occurs.
V_WAITALL	Waits until the completion of all I/O regardless of any occurrence of I/O errors.

Execution Environment

The **vms_iowait** and **vms_iowaitf** kernel services can be called from the process environment only.

They can only be used by file segments.

Return Values

Item	Description
0	Indicates that the page-out operations completed.
EIO	Indicates that an error occurred while performing I/O.

Related reference:

"vm_invalidatep Kernel Service" on page 543

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vm_uiomove Kernel Service

Purpose

Moves data between a virtual memory object and a buffer specified in the uio structure.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

```
#include <sys/uio.h>

int vm_uiomove (vmid, limit, rw, uio)
vmid_t vmid;
int limit;
enum uio_rw rw;
struct uio *uio;
```

Parameters

Item	Description
vmid	Specifies the virtual memory object ID.
limit	Indicates the limit on the transfer length. If this parameter is negative or 0, no bytes are transferred.
rw	Specifies a read/write flag that gives the direction of the move. The possible values for this parameter (**UIO_READ**, **UIO_WRITE**) are defined in the **/usr/include/sys/uio.h** file.
uio	Points to the **uio** structure.

Description

The **vm_uiomove** kernel service moves data between a virtual memory object and the buffer specified in a uio structure.

This service determines the virtual addressing required for the data movement according to the offset in the object.

The **vm_uiomove** kernel service is similar to the **uiomove** kernel service, but the address for the trusted buffer is specified by the *vmid* parameter and the uio_offset field of *offset* parameters instead of as a **caddr_t** address. The offset size is a 64 bit offset_t, which allows file offsets in client segments which are greater than 2 gigabytes. **vm_uiomove** must be used instead of **vm_move** if the client filesystem supports files which are greater than 2 gigabytes.

Note: The **vm_uiomove** kernel service does not support use of cross-memory descriptors.

I/O errors for paging space and a lack of paging space are reported as signals.

Execution Environment

The **vm_uiomove** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.
EFAULT	Indicates a bad address.
ENOMEM	Indicates insufficient memory.
ENOSPC	Indicates insufficient disk space.
EIO	Indicates an I/O error.

Other file system-specific **errno** global variables are returned by the virtual file system involved in the move function.

Related reference:

"uiomove Kernel Service" on page 516

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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,  devid)
int type;
dev_t devid)();
```

Parameters

Item	Description
type	Specifies the type of device. You can specify multiple values. But the *type* parameter must have a value of **D_REMOTE** as one of its values. You can also specify the following optional value: **D_NOWAIT** Indicates that if I/O discovered during a prior **vm_setdevid** call has not yet completed, the paging device table entry will be removed, asynchronously, at a future point in time when all such I/O to it has completed. This particular **vm_umount** kernel service call will return without waiting for the I/O to complete. Any **buf** structures associated with this paging device entry remain allocated until the paging device entry is finally removed.
devid	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

Item	Description
0	Indicates successful completion.
EINVAL	Indicates that a file system with the strategy routine designated by the *devid* parameter is not in the paging device table.

Related reference:

"vm_mount Kernel Service" on page 548

"vm_setdevid Kernel Service" on page 567

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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;
```

Description

The **vm_write** kernel service initiates page-out for pages that intersect the address range *(vaddr*, *vaddr* + *nbytes*).

If the *force* parameter is nonzero, modified pages are written to disk regardless of how recently they have been written.

Page-out is initiated for each modified page. An unchanged page is left in memory with its reference bit set to 0. This makes the unchanged page a candidate for the page replacement algorithm.

The caller must have write access to the specified pages.

The initiated I/O is asynchronous. The **vms_iowait** kernel service can be called to wait for I/O completion.

Note: The **vm_write** subroutine is not supported for use on large pages.

Parameters

Item	Description
vaddr	Specifies the address of the first byte of the page range for which a page-out is desired.
nbytes	Specifies the number of bytes starting at the byte specified by the *vaddr* parameter. This parameter must be nonnegative. All of the bytes must be in the same virtual memory object.
force	Specifies a flag indicating that a modified page is to be written regardless of when it was last written.

Execution Environment

The **vm_write** kernel service can be called from the process environment only.

Return Values

Item	Description
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. • One or more large pages lie in the target page range.
EACCES	Indicates that access does not permit writing.
EIO	Indicates a permanent I/O error.

Related reference:

"vm_writep Kernel Service" on page 578

"vms_iowait, vms_iowaitf Kernel Services" on page 573

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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;
```

Description

The **vm_writep** kernel service initiates page-out for the specified page range in the virtual memory object. I/O is initiated for modified pages only. Unchanged pages are left in memory, but their reference bits are set to 0.

The caller can wait for the completion of I/O initiated by this and prior calls by calling the **vms_iowait** kernel service.

Note: The **vm_writep** subroutine is not supported for use on large pages.

Parameters

Item	Description
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.

Execution Environment

The **vm_writep** kernel service can be called from the process environment only.

Return Values

Item **Description**

0 Indicates successful completion.

EINVAL Indicates any one of the following errors:

- *pfirst* = 0 and *npages* = 0.
- The virtual memory object ID is not valid.
- The starting page is negative.
- The number of pages is negative.
- The page range exceeds the size of virtual memory object.
- One or more large pages lie in the target page range.

Related reference:

“vm_invalidatep Kernel Service” on page 543 "vm_write Kernel Service" on page 576

"vms_iowait, vms_iowaitf Kernel Services" on page 573

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

vn_free Kernel Service

Purpose

Frees a v-node previously allocated by the **vn_get** kernel service.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>

int vn_free ( vp)
struct vnode *vp;
```

Parameter

Item	Description
vp	Points to the v-node to be deallocated.

Description

The **vn_free** kernel service provides a mechanism for deallocating v-node objects used within the virtual file system. The v-node specified by the *vp* parameter is returned to the pool of available v-nodes to be used again.

Execution Environment

The **vn_free** kernel service can be called from the process environment only.

Return Values

The **vn_free** service always returns 0.

Related reference:

"vn_get Kernel Service"

Related information:

Virtual File System Overview

Virtual File System (VFS) Kernel Services

vn_get Kernel Service

Purpose

Allocates a virtual node.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
```

```
int vn_get ( vfsp,  gnp,  vpp)
struct vfs *vfsp;
struct gnode *gnp;
struct vnode **vpp;
```

Parameters

Item	Description
vfsp	Points to a **vfs** structure describing the virtual file system that is to contain the v-node. Any returned v-node belongs to this virtual file system.
gnp	Points to the g-node for the object. This pointer is stored in the returned v-node. The new v-node is added to the list of v-nodes in the g-node.
vpp	Points to the place in which to return the v-node pointer. This is set by the **vn_get** kernel service to point to the newly allocated v-node.

Description

The **vn_get** kernel service provides a mechanism for allocating v-node objects for use within the virtual file system environment. A v-node is first allocated from an effectively infinite pool of available v-nodes.

Upon successful return from the **vn_get** kernel service, the pointer to the v-node pointer provided (specified by the *vpp* parameter) has been set to the address of the newly allocated v-node.

The fields in this v-node have been initialized as follows:

Field	Initial Value
`v_count`	Set to 1.
`v_vfsp`	Set to the value in the *vfsp* parameter.
`v_gnode`	Set to the value in the *gnp* parameter.
`v_next`	Set to list of others v-nodes with the same g-node.

All other fields in the v-node are zeroed.

Execution Environment

The **vn_get** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates successful completion.
ENOMEM	Indicates that the **vn_get** kernel service could not allocate memory for the v-node. (This is a highly unlikely occurrence.)

Related reference:

"vn_free Kernel Service" on page 579

Related information:

Virtual File System Overview

Virtual File System (VFS) Kernel Services

vsx_disable Kernel Service

Purpose

Communicates the status of the vector and the vector-scalar registers to the hypervisor.

Syntax

```
#include <sys/machine.h>
void vsx_disable (old)
char old;
```

Parameters

old
    Specifies the value returned by the vsx_enable kernel service.

Description

The vsx_disable kernel service communicates to the hypervisor that the vector and the vector-scalar registers are no longer in use.

Execution Environment

The vsx_disable kernel service can be called from the process environment or the interrupt environment. The vsx_disable kernel service must be called while all interrupts from within the INTMAX critical section are disabled.

Return Values

The vsx_disable kernel service has no return values.

vsx_enable Kernel Service

Purpose

Communicates the status of the vector and the vector-scalar registers to the hypervisor.

Syntax

```
#include <sys/machine.h>
char vsx_enable ()
```

Description

The vsx_enable kernel service communicates to the hypervisor that the vector and the vector-scalar registers are in use.

Execution Environment

The vsx_enable kernel service can be called from the process environment or the interrupt environment. The vsx_enable kernel service must be called while all the interrupts from within the INTMAX critical section are disabled.

Return Values

The vsx_enable kernel service returns the current setting.

W

The following kernel services begin with the with the letter w.

waitcfree Kernel Service

Purpose

Checks the availability of a free character buffer.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/cblock.h>
#include <sys/sleep.h>
```

```
int waitcfree ( )
```

Description

The **waitcfree** kernel service is used to wait for a buffer which was allocated by a previous call to the **pincf** kernel service. If one is not available, the **waitcfree** kernel service waits until either a character buffer becomes available or a signal is received.

The **waitcfree** kernel service has no parameters.

Execution Environment

The **waitfree** kernel service can be called from the process environment only.

Return Values

Item	Description
EVENT_SUCC	Indicates a successful operation.
EVENT_SIG	Indicates that the wait was terminated by a signal.

Related reference:

"pincf Kernel Service" on page 409

"putcf Kernel Service" on page 425

Related information:

I/O Kernel Services

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

Item	Description
queue_id	Specifies the device queue identifier.

Description

The **waitq** kernel service is not part of the base kernel but is provided by the device queue management kernel extension. This queue management kernel extension must be loaded into the kernel before loading any kernel extensions referencing these services.

The **waitq** kernel service waits for a queue element to be placed on the device queue specified by the *queue_id* parameter. This service performs these two actions:

- Waits on the event mask associated with the device queue.
- Calls the **readq** kernel service to make the most favored queue element the active one.

Processes can only use the **waitq** kernel service to wait for a single device queue. Use the **et_wait** service to wait on the occurrence of more than one event, such as multiple device queues.

The **waitq** kernel service uses the **EVENT_SHORT** form of the **et_wait** kernel service. Therefore, a signal does not terminate the wait. Use the **et _wait** kernel service if you want a signal to terminate the wait.

The **readq** kernel service can be used to read the active queue element from a queue. It does not wait for a queue element if there are none in the queue.

> **Attention:** The server must not alter any fields in the queue element or the system may halt.

Execution Environment

The **waitq** kernel service can be called from the process environment only.

Return Values

The **waitq** service returns the address of the active queue element in the device queue.

Related reference:

"et_wait Kernel Service" on page 119

WPAR_CKPT_QUERY (Checkpoint Query) Device Driver ioctl Operation

Purpose

Queries a device driver about its checkpoint capabilities.

Syntax

```
#include <sys/ioctl.h>
```

int ioctl (*FileDescriptor*, WPAR_CKPT_QUERY, *Arg* **)**
int *FileDescriptor*;
wpar_ckpt_resp_t * *Arg*;

Parameters

Item	Description
FileDescriptor	Open file descriptor that refers to the device being queried for the checkpoint capability.
WPAR_CKPT_QUERY	The command that requests information on the device checkpoint capability.
Arg	Pointer to a **wpar_ckpt_resp_t** structure which will contain a device driver response on the checkpoint capability upon a successful return from the **ioctl** call.

Description

The **WPAR_CKPT_QUERY** operation allows a caller to ask a device driver connected to the ioctl input file descriptor if it supports checkpoint and restart operations. If a device driver supports checkpoint and restart operations, the returned answer can describe what operations are required to accomplish a checkpoint and restart.

If the device is not checkpoint and restart capable, checkpoint-aware devices fail this ioctl request with the **ENOSYS** error. Non-checkpoint-aware devices fail this ioctl request as an unknown ioctl. If the device is checkpoint and restart capable, checkpoint-aware devices return success.

The *arg* parameter to a **WPAR_CKPT_QUERY** ioctl request allows the caller to receive specific information regarding how the device supports checkpoint and restart if it is capable. The caller of a **WPAR_CKPT_QUERY** ioctl request must supply a pointer to a structure of the **wpar_ckpt_resp_t** type in the *arg* parameter.

wpar_ckpt_resp_t structure

The **wpar_ckpt_resp_t** structure is supplied as the input to the **WPAR_CKPT_QUERY** ioctl request.

```
#define WPAR_CKPT_OP_MAX 5
typedef struct wpar_ckpt_resp_t {
int opcnt;
wpar_ckpt_op_top [WPAR_CKPT_OP_MAX];
}wpar_ckpt_resp_t;
```

The fields of the **wpar_ckpt_resp_t** structure are as follows:

Item	Description
opcnt	Returned from an **WPAR_CKPT_QUERY** ioctl request as the number of the **wpar_ckpt_op_t** sub-structures that contain return information.
wpar_ckpt_op_top	A sub-structure that contains specific information on operation types that must occur on a device for it to save or restore its state correctly.

wpar_ckpt_op_t structure

The **wpar_ckpt_op_t** structure is a sub-structure of the **wpar_ckpt_resp_t** structure.

```
typedef struct wpar_ckpt_op_t {
int op;
int opt; /*extended options of openx*/
}wpar_ckpt_op_t;
```

The fields of the **wpar_ckpt_op_t** structure are as follows:

Item	Description
op	Returned from a **WPAR_CKPT_QUERY** ioctl request. Defined as a set of one or more operations that must be performed to successfully checkpoint and restart the device.
opt	Options to supply to the **openx** function if the device is to be re-opened on the arrival server through the **openx** function.

wpar_ckpt_op_t op field

Item	Description
WPAR_CKPT_OP_NULL	Device requires no special handling for checkpoint and restart operations.
WPAR_CKPT_OP_REOPEN	Device needs to be re-opened through the **open** function with the access modes applicable at checkpoint time.
WPAR_CKPT_OP_OPENX	Device needs to be re-opened with the **openx** function. The *opt* field denotes the desired extension argument to the **openx** function.

Return Values

Upon successful completion, this operation returns a value of 0. Otherwise, it returns a value of -1 and the errno global variable is set to one of the following values:

Item	Description
ENOSYS	Device cannot participate in checkpoint and restart operations.
EINVAL	Device does not accept the **WPAR_CKPT_QUERY** operation.

Related reference:

"kwpar_checkpoint_status Kernel Service" on page 313

w_clear Kernel Service

Purpose

Removes a watchdog timer from the list of watchdog timers known to the kernel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/watchdog.h>

int w_clear ( w)
struct watchdog *w;
```

Parameter

Item	Description
w	Specifies the watchdog timer structure.

Description

The watchdog timer services, including the **w_clear** kernel service, are typically used to verify that an I/O operation completes in a reasonable time.

When the **w_clear** kernel service removes the watchdog timer, the *w*->**count** watchdog count is no longer decremented. In addition, the *w*->**func** watchdog timer function is no longer called.

In a uniprocessor environment, the call always succeeds. This is untrue in a multiprocessor environment, where the call will fail if the watchdog timer is being handled by another processor. Therefore, the function now has a return value, which is set to 0 if successful, or -1 otherwise. Funnelled device drivers do not need to check the return value since they run in a logical uniprocessor environment. Multiprocessor-safe and multiprocessor-efficient device drivers need to check the return value in a loop. In addition, if a driver uses locking, it must release and reacquire its lock within this loop, as shown below:

```
while (w_clear(&watchdog))
  release_then_reacquire_dd_lock;
                  /* null statement if locks not used */
```

Note: The **w_clear** kernel service clears any attributes that were previously set by using the **w_setattr()** kernel service.

Execution Environment

The **w_clear** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates that the watchdog timer was successfully removed.
-1	Indicates that the watchdog timer could not be removed.

Related reference:

"w_init Kernel Service"

"w_setattr Kernel Service" on page 587

Related information:

Timer and Time-of-Day Kernel Services

w_init Kernel Service

Purpose

Registers a watchdog timer with the kernel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/watchdog.h>

int w_init ( w)
struct watchdog *w;
```

Parameter

Item	Description
w	Specifies the watchdog timer structure.

Description

The **watchdog** structure must be initialized prior to calling the **w_init** kernel service as follows:

- Set the `next` and `prev` fields to NULL.
- Set the `func` and `restart` fields to the appropriate values.
- Set the `count` field to 0.

> **Attention:** The watchdog structure must be pinned when the **w_init** service is called. It must remain pinned until after the call to the **w_clear** service. During this time, the watchdog structure must not be altered except by the watchdog services.

The watchdog timer services, including the **w_init** kernel service, are typically used to verify that an I/O operation completes in a reasonable time. The watchdog timer is initialized to the stopped state and must be started using the **w_start** service.

In both uniprocessor and multiprocessor environments, the **w_init** kernel service always succeeds.

The calling parameters for the watchdog timer function are:

```
void func (w)
struct watchdog *w;
```

Execution Environment

The **w_init** kernel service can be called from the process environment only.

Return Values

The **w_init** kernel service returns 0 for compatibility with previous releases of AIX.

Related reference:

"w_clear Kernel Service" on page 585

"w_setattr Kernel Service"

Related information:

Timer and Time-of-Day Kernel Services

w_setattr Kernel Service

Purpose

Sets attributes for a watchdog timer.

Syntax

```
#include <sys/watchdog.h>
#include <sys/kerrno.h>

kerrno_t w_setattr(struct watchdog *w, char attr)
```

Parameter

Item	Description
w	Specifies the watchdog timer structure.
attr	A bitmask of attributes to be set. Supported flags are: **WD_ATTR_MOVE_OK** Allow timer to migrate from one CPU to another.

Description

The **w_setattr** kernel service sets attributes for the specified watchdog timer. The **WD_ATTR_MOVE_OK** attribute should be set when the caller does not have a dependency on which processor the timer expiration handler is called. This attribute allows the system to move the timer from one processor to another as needed, to improve the effectiveness of processor folding. When this attribute is set, the associated watchdog timer is moved to another processor when the owning processor is folded.

The **w_setattr** kernel service must be called after the **w_init()** kernel service but before the **w_start()** kernel service. Otherwise, the **w_setattr** kernel service may fail.

Execution Environment

The **w_setattr** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
0	The specified attribute was successfully set.
<0	The specified attribute was not set. The failure is indicated with return value set to one of the following values: EINVAL_W_SETATTR_EYEC: An invalid eye catcher was detected. EINVAL_W_SETATTR_ATTR: An invalid attribute flag was detected.

Related reference:
"w_clear Kernel Service" on page 585
"w_start Kernel Service"
"w_stop Kernel Service" on page 589

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

Item	Description
w	Specifies the watchdog timer structure.

Description

The watchdog timers, including the **w_start** kernel service, are typically used to verify that an I/O operation completes in a reasonable time. The **w_start** and **w_stop** kernel services are designed to allow the timer to be started and stopped efficiently. The kernel decrements the *w*->**count** watchdog count every second. The kernel calls the *w*->**func** watchdog timer function when the *w*->**count** watchdog count reaches 0. A watchdog timer is ignored when the *w*->**count** watchdog count is less than or equal to 0.

The **w_start** kernel service sets the *w*->**count** watchdog count to a value of *w*->**restart**.

> **Attention:** The watchdog structure must be pinned when the **w_start** kernel service is called. It must remain pinned until after the call to the **w_clear** kernel service. During this time, the watchdog structure must not be altered except by the watchdog services.

Execution Environment

The **w_start** kernel service can be called from the process and interrupt environments.

Return Values

The **w_start** kernel service has no return values.

Related reference:

"w_stop Kernel Service"

"w_setattr Kernel Service" on page 587

Related information:

Timer and Time-of-Day Kernel Services

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

Item	Description
w	Specifies the watchdog timer structure.

Description

The watchdog timer services, including the **w_stop** kernel service, are typically used to verify that an I/O operation completes in a reasonable time. The **w_start** and **w_stop** kernel services are designed to allow the timer to be started and stopped efficiently. The kernel decrements the *w*->**count** watchdog count every second. The kernel calls the *w*->**func** watchdog timer function when the *w*->**count** watchdog count reaches 0. A watchdog timer is ignored when *w*->**count** is less than or equal to 0.

> **Attention:** The watchdog structure must be pinned when the **w_stop** kernel service is called. It must remain pinned until after the call to the **w_clear** kernel service. During this time, the watchdog structure must not be altered except by the watchdog services.

Execution Environment

The **w_stop** kernel service can be called from the process and interrupt environments.

Return Values

The **w_stop** kernel service has no return values.

Related reference:

"w_start Kernel Service" on page 588

"w_setattr Kernel Service" on page 587

Related information:

Timer and Time-of-Day Kernel Services

x

The following kernel services begin with the with the letter x.

xfidToName() Kernel Service

Purpose

Finds the full path name of the file corresponding to an `xfid_t` structure.

Syntax

```
#include <sys/xfops.h>

int     xfidToName(struct xfid *xfp,
                          void *nrp,
                          char *pathname,
                          unsigned int pbuflen,
                          long flags);
```

Description

The `xfidToName()` kernel service finds a name for an `xfid` value.

Parameters

`xfp`
Pointer to the `xfid` value for which a name is needed.

`nrp`
Name resolution structure that is passed to the validation routine.

`pathname`
Pointer to buffer where the file name will be stored.

`pbuflen`
Size of path name buffer. A size of `MAXPATHLEN` is sufficient to hold any path name.

`flags`
Operation modifiers. This parameter must be set to zero.

Return values

0 Indicates success. The path name for the `xfid` value is returned.

ENOENT
Name not found.

EPERM
No permission for lookup.

EINVAL
Invalid parameter is specified.

E2BIG Path is larger than `pbuflen` bytes.

xlate_create Kernel Service

Purpose

Creates pretranslation data structures.

Syntax

```
int xlate_create (dp, baddr, count, flags)
struct xmem*dp;
caddr_t baddr;
int count;
uint flags;
```

Description

The **xlate_create** kernel service creates pretranslation data structures capable of pretranslating all pages of the virtual buffer indicated by the *baddr* parameter for length of *count* into a list of physical page numbers, appended to the cross memory descriptor pointed to by *dp*.

If the **XLATE_ALLOC** flag is set, only the data structures are created and no pretranslation is done. If the flag is not set, in addition to the data structures being created, each page of the buffer is translated and the access permissions verified, requiring read-write access to each page. The **XLATE_ALLOC** flag is useful when the buffer will be pinned and utilized later, through the **xlate_pin** and **xlate_unpin** kernel services.

The **XLATE_SPARSE** flag can be used to indicate that only selected portions of a pretranslated region may be valid (pinned and pretranslated) at any given time. The **XLATE_SPARSE** flag can be used in conjunction with the **XLATE_ALLOC** flag to preallocate the pretranslation data structures for an address region that will be dynamically managed.

The **xlate_create** kernel service is primarily for use when memory buffers will be reused for I/O. The use of this service to create a pretranslation for the memory buffer avoids page translation and access checking overhead for all future DMAs involving the memory buffer until the **xlate_remove** kernel service is called.

Parameters

Item **Description**

dp Points to the cross memory descriptor.

baddr Points to the virtual buffer.

count Specifies the length of the virtual buffer.

flags Specifies the operation. Valid values are as follows:

XLATE_PERSISTENT
Indicates that the pretranslation data structures should be persistent across calls to pretranslation services.

XLATE_ALLOC
Indicates that the pretranslation data structures should be allocated only, and no translation should be performed.

XLATE_SPARSE
Indicates that the pretranslation information will be sparse, allowing for the coexistence of valid (active) pretranslation regions and invalid (inactive) pretranslation regions.

Return Values

Item	Description
ENOMEM	Unable to allocate memory
XMEM_FAIL	No physical translation, or No Access to a Page
XMEM_SUCC	Successful pretranslation created

Execution Environment

The **xlate_create** kernel service can only be called from the process environment. The entire buffer must be pinned (unless the **XLATE_ALLOC** flag is set), and the cross memory descriptor valid.

Related reference:

"xlate_remove Kernel Service" on page 593

"xlate_pin Kernel Service"

"xlate_unpin Kernel Service" on page 594

xlate_pin Kernel Service

Purpose

Pins all pages of a virtual buffer.

Syntax

```
int xlate_pin (dp, baddr, count, rw)
struct xmem *dp;
caddr_t baddr;
int count;
int rw;
```

Description

The **xlate_pin** kernel service pins all pages of the virtual buffer indicated by the *baddr* parameter for length of *count* and also appends pretranslation information to the cross memory descriptor pointed to by the *dp* parameter.

The **xlate_pin** kernel service results in a short-term pin, which will support **mmap** and **shmatt** allocated memory buffers.

In addition to pinning and translating each page, the access permissions to the page are verified according to the desired access (as specified by the *rw* parameter). For a setting of **B_READ**, write access to the page must be allowed. For a setting of **B_WRITE**, only read access to the page must be allowed.

The caller can preallocate pretranslation data structures and append them to the cross memory descriptor prior to the call (through a call to the **xlate_create** kernel service), or have this service allocate the necessary data structures. If the cross memory descriptor is already of type **XMEM_XLATE**, it is assumed that the data structures are already allocated. If callers want to have the pretranslation data structures persist across the subsequent **xlate_unpin** call, they should also set the **XLATE_PERSISTENT** flag on the call to the **xlate_create** kernel service.

Parameters

Item	Description
dp	Points to the cross memory descriptor.
baddr	Points to the virtual buffer.
count	Specifies the length of the virtual buffer.
rw	Specifies the access permissions for each page.

Return Values

If successful, the **xlate_pin** kernel service returns 0. If unsuccessful, one of the following is returned:

Item	Description
EINVAL	Invalid cross memory descriptor or parameters.
ENOMEM	Unable to allocate memory.
ENOSPC	Out of Paging Resources.
XMEM_FAIL	Page Access violation.

Execution Environment

The **xlate_pin** kernel service is only callable from the process environment, and the cross memory descriptor must be valid.

Related reference:

"xm_det Kernel Service" on page 595

"xm_mapin Kernel Service" on page 595

"xlate_unpin Kernel Service" on page 594

xlate_remove Kernel Service

Purpose

Removes physical translation information from an xmem descriptor from a prior **xlate_create** call.

Syntax

```
caddr_t xlate_remove (dp)
struct xmem *dp;
```

Description

See the **xlate_create** kernel service.

Parameters

Item	Description
dp	Points to the cross memory descriptor.

Return Values

Item	Description
XMEM_FAIL	No pretranslation information present in the xmem descriptor.
XMEM_SUCC	Pretranslation successfully removed.

Execution Environment

The **xlate_remove** kernel service can only be called from the process environment.

Related reference:

"xm_det Kernel Service" on page 595

"xlate_pin Kernel Service" on page 592

"xlate_unpin Kernel Service"

xlate_unpin Kernel Service

Purpose

Unpins all pages of a virtual buffer.

Syntax

```
int xlate_unpin (dp, baddr, count)
struct xmem *dp;
caddr_t baddr;
int count;
```

Description

The **xlate_unpin** kernel service unpins pages from a prior call to the **xlate_pin** kernel service based on the *baddr* and *count* parameters. It does this by utilizing the pretranslated real page numbers appended to the cross memory descriptor pointed to by *dp*.

If the **XLATE_PERSISTENT** flag is not set in the **prexflags** flag word of the pretranslation data structure, the pretranslation data structures are also freed.

Parameters

Item	Description
dp	Points to the cross memory descriptor.
baddr	Points to the virtual buffer.
count	Specifies the length of the virtual buffer.

Return Values

If successful, the **xlate_unpin** kernel service returns 0. If unsuccessful, one of the following is returned:

Item	Description
EINVAL	Invalid cross memory descriptor or parameters.
ENOSPC	Unable to allocate paging space (case of **mmap** segment).
ENOSPC	Out of Paging Resources.
XMEM_FAIL	Page Access violation.

Related reference:

"xm_det Kernel Service" on page 595

"xm_mapin Kernel Service" on page 595

"xlate_pin Kernel Service" on page 592

xm_det Kernel Service

Purpose

Releases the addressability to the address space described by an xmem descriptor.

Syntax

```
void xm_det (baddr, dp)
caddr_t baddr;
struct xmem *dp;
```

Description

See the **xm_mapin** Kernel Service for more information.

Parameters

Item	Description
baddr	Specifies the effective address previously returned from the **xm_mapin** kernel service.
dp	Cross memory descriptor that describes the above memory object.

Related reference:
"xlate_create Kernel Service" on page 590
"xlate_remove Kernel Service" on page 593
"xm_mapin Kernel Service"

xm_mapin Kernel Service

Purpose

Sets up addressability in the current process context.

Syntax

```
#include <sys/adspace.h>int xm_mapin (dp, baddr, count, eaddr)
struct xmem *dp;caddr_t baddr;
size_t count;
caddr_t *eaddr;
```

Description

The **xm_mapin** kernel service sets up addressability in the current process context to the address space indicated by the cross memory descriptor pointed to by the *dp* parameter for the addresses [*baddr*, *baddr* + *count* - 1].

This service is created specifically for Client File Systems, or others who need to setup addressability to an address space defined by an xmem descriptor.

If the requested mapping spans a segment boundary, no mapping will be performed, and a return code of **EAGAIN** is returned to indicate that individual calls to the **xm_mapin** kernel service are necessary to map the portions of the buffer in each segment. The **xm_mapin** kernel service must be called again with the original *baddr* and a *count* indicating the number of bytes to the next segment. (The number of bytes to the next segment boundary can be obtained using the **xm_maxmap** kernel service.) This will provide an effective address to use for accessing this portion of the buffer. Then, iteratively, **xm_mapin** must be called with the segment boundary address (previous *baddr* + *count*), and a new *count* indicating the remainder of the buffer or the next segment boundary, whichever is smaller. This will provide another effective address to use for accessing the next portion of the buffer.

Each address set up by the **xm_mapin** kernel service must be undone with the **xm_det** kernel service when it is no longer needed because the **xm_mapin** kernel service currently uses the **vm_att** kernel service.

Parameters

Item	Description
dp	Points to the cross memory descriptor.
baddr	Points to the virtual buffer.
count	Specifies the length of the virtual buffer to map.
eaddr	Points to where the effective address to access the data buffer is returned.

Return Values

Item	Description
0	Successful. (Reference Parameter *eaddr* contains the address to use)
XMEM_FAIL	Invalid cross memory descriptor.
EAGAIN	Segment boundary crossing encountered. Caller should make separate **xm_mapin** calls to map each segments worth.

Execution Environment

The **xm_mapin** kernel service can be called from the process or interrupt environments.

Related reference:

"xm_det Kernel Service" on page 595

"xlate_remove Kernel Service" on page 593

"xlate_pin Kernel Service" on page 592

"xm_maxmap Kernel Service"

xm_maxmap Kernel Service

Purpose

Determines the maximum permissible count value for a subsequent call to **xm_mapin**.

Syntax

```
#include <sys/adspace.h>

int xm_maxmap (dp, uaddr, len)
 struct xmem *dp;
 void *uaddr;
 size_t *len;
```

Parameters

Item	Description
dp	Points to the cross memory descriptor.
uaddr	Points to the virtual buffer.
len	Points to where the maximum permissible count is returned.

Description

The **xm_maxmap** kernel service determines the maximum permissible count value (in bytes) for a subsequent **xm_mapin** call. The value is determined based on the input cross-memory descriptor *dp* and the starting address *uaddr*, and it is returned in the *len* parameter. There is no guarantee that **xm_mapin** will succeed; however, it is guaranteed that *uaddr + *len* - 1 is in the same segment as *uaddr*, and therefore **xm_mapin** will not return **EAGAIN**.

Execution Environment

The **xm_maxmap** interface can be called from the process or interrupt environment.

Return Values

Item	Description
XMEM_SUCC	Successful (Reference parameter *len* contains the maximum permissible value for a subsequent **xm_mapin** call)
XMEM_FAIL	Invalid cross memory descriptor.
EAGAIN	Segment boundary crossing encountered. Caller should make separate **xm_mapin** calls to map each segment's worth.

Related reference:
"xm_mapin Kernel Service" on page 595

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

Item	Description
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.

There are multiple heaps provided by the kernel for use by kernel extensions. Two primary kernel heaps are **kernel_heap** and **pinned_heap**. Kernel extensions should use the **kernel_heap** value when allocating memory that is not pinned, and should use the **pinned_heap** value when allocating memory that should always be pinned or pinned for long periods of time. When allocating from the **pinned_heap** heap, the **xmalloc** kernel service will pin the memory before a successful return. The **pin** and **unpin** kernel services should be used to pin and unpin memory from the **kernel_heap** heap when the memory should only be pinned for a limited amount of time. Memory from the **kernel_heap** heap must be unpinned before freeing it. Memory from the **pinned_heap** heap should not be unpinned.

The **kernel_heap** heap points to one of the following heaps: **kernel_heap_4K_64K** and **kernel_heap_16M**. The **pinned_heap** heap points to one of the following heaps: **pinned_heap_4K_64K** and **pinned_heap_16M**. Each of the target heaps differ in the size of the pages that back them. **kernel_heap_4K_64K** or **pinned_heap_4K_64K** will be backed by either medium (64 KB) or regular (4 KB) pages, depending on the page size supported by the machine. **kernel_heap_16M** or **pinned_heap_16M** will return memory backed by large pages if large page heaps are enabled. If large page heaps are not enabled, **kernel_heap** or **pinned_heap** will point to the default heap. If the size of the backing pages are not important, use the **kernel_heap** value and the **pinned_heap** value. They will point to the heap that you prefer. For more information about large page heap support, see **vmo**.

Kernel extensions can use these services to allocate memory out of the kernel heaps. For example, the **xmalloc** (**128**,**3**,**kernel_heap**) kernel service allocates a 128-byte double word aligned area out of the kernel heap.

A kernel extension must use the **xmfree** kernel service to free the allocated memory. If it does not, subsequent allocations eventually are unsuccessful.

The **xmalloc** kernel service has two compatibility interfaces: **malloc** and **palloc**.

The following additional interfaces to the **xmalloc** kernel service are provided:

- **malloc** (*size*) is equivalent to **xmalloc** (*size*, **0**, **kernel_heap**).
- **palloc** (*size*, *align*) is equivalent to **xmalloc** (*size*, *align*, **kernel_heap**).

Execution Environment

The **xmalloc** kernel service can be called from the process environment only.

Return Values

Upon successful completion, the **xmalloc** kernel service returns the address of the allocated area. A null pointer is returned under the following circumstances:

- The requested memory cannot be allocated.
- The heap has not been initialized for memory allocation.

Related reference:

"xmfree Kernel Service" on page 610

Related information:

Memory Kernel Services

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

Item	**Description**
addr	Specifies the address of the user buffer to be accessed in a cross-memory operation.
count	Indicates the size of the user buffer to be accessed in a cross-memory operation.
dp	Specifies a cross-memory descriptor. The *dp*->**aspace_id** variable must be set to a value of **XMEM_INVAL**.
segflag	Specifies a segment flag. This flag is used to determine the address space of the memory that the cross-memory descriptor applies to, as well as for other purposes. The valid values for this flag can be found in the **/usr/include/xmem.h** file.

Description

The **xmattach** kernel service prepares the user buffer so that a device driver can access it without executing under the process that requested the I/O operation. A device top-half routine calls the **xmattach** kernel service. The **xmattach** kernel service allows a kernel process or device bottom-half routine to access the user buffer with the **xmemin** or **xmemout** kernel services. The device driver must use the **xmdetach** kernel service to inform the kernel when it has finished accessing the user buffer.

The kernel remembers which segments are attached for cross-memory operations. Resources associated with these segments cannot be freed until all cross-memory descriptors have been detached. "Cross Memory Kernel Services" in Memory Kernel Services in in *Kernel Extensions and Device Support Programming Concepts* describes how the cross-memory kernel services use cross-memory descriptors.

Note: When the **xmattach** kernel service remaps user memory containing the cross-memory buffer, the effects are machine-dependent. Also, cross-memory descriptors are not inherited by a child process.

Storage-key protection can be enforced on memory regions described by a cross-memory descriptor. The enforcement is done during normal access checking performed by cross-memory services, such as the **xmemdma** kernel service. A kernel keyset can be contained in the cross-memory descriptor to limit memory accessibility. When a keyset is associated with a cross-memory descriptor, access to the memory region is limited by that keyset. A keyset is required because a cross-memory descriptor can describe a virtual memory region with multiple keys assigned to the pages it contains. Normally, a keyset describes the accessibility of the context that the attach was initiated for. For example, a cross-memory attached to user-space contains a description of the user-mode accessibility (keyset). Adding keysets to kernel cross-memory descriptors can also enhance system RAS, since they limit kernel access by the cross-memory descriptor. Typically it is limited to that of the **xmattach** caller or to specific key(s), to catch cases where a cross-memory descriptor is misused.

User-mode storage-keys are always associated with descriptors attached using **USER_SPACE** or **USERI_SPACE** segflag. These flags were always required to attach to the user address space, so no explicit update is required to enable storage-key protection on user memory attaches. Once attached, existing kernel services that require cross-memory descriptors enforce the user keyset saved at attach time when performing memory accesses or checking user accessibility.

For kernel memory, a keyset is not used to restrict regions attached with **SYS_ADSPACE**. Attaching a region with **SYS_ADSPACE_ASSIGN_KEYSET** associates the caller's keyset with the cross-memory region.

Execution Environment

The **xmattach** kernel service can be called from the process environment only.

Return Values

Item	Description
XMEM_SUCC	Indicates a successful operation.
XMEM_FAIL	Indicates one of the following errors: • The buffer size indicated by the *count* parameter is less than or equal to 0. • The cross-memory descriptor is in use (*dp*->**aspace_id != XMEM_INVAL**). • The area of memory indicated by the *addr* and *count* parameters is not defined.

Related reference:

"uphysio Kernel Service" on page 523

"xmdetach Kernel Service"

"xmgethkeyset Kernel Service" on page 611

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

Item	Description
dp	Points to a cross-memory descriptor initialized by the **xmattach** kernel service.

Description

The **xmdetach** kernel service informs the kernel that a user buffer can no longer be accessed. This means that some previous caller, typically a device driver bottom half or a kernel process, is no longer permitted to do cross-memory operations on this buffer. Subsequent calls to either the **xmemin** or **xmemout** kernel service using this cross-memory descriptor result in an error return. The cross-memory descriptor is set to *dp*->**aspace_id = XMEM_INVAL** so that the descriptor can be used again. "Cross Memory Kernel Services" in Memory Kernel Services in *Kernel Extensions and Device Support Programming Concepts* describes how the cross-memory kernel services use cross-memory descriptors.

Execution Environment

The **xmdetach** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
XMEM_SUCC	Indicates successful completion.
XMEM_FAIL	Indicates that the descriptor was not valid or the buffer was not defined.

Related reference:

"xmattach Kernel Service" on page 598

"xmemout Kernel Service" on page 608

Related information:

Cross Memory Kernel Services

xmemdma Kernel Service

Purpose

Prepares a page for direct memory access (DMA) I/O or processes a page after DMA I/O is complete.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmem.h>

int xmemdma ( xp,  xaddr,  flag)
struct xmem *xp;
caddr_t xaddr;
int flag;
```

Parameters

Item	Description
xp	Specifies a cross-memory descriptor.
xaddr	Identifies the address specifying the page for transfer.
flag	Specifies whether to prepare a page for DMA I/O or process it after DMA I/O is complete. Possible values are: **XMEM_ACC_CHK** Performs access checking on the page. When this flag is set, the page protection attributes are verified. **XMEM_DR_SAFE** Indicates that the use of the real memory address is DLPAR safe. **XMEM_HIDE** Prepares the page for DMA I/O. For cache-inconsistent platforms, this preparation includes hiding the page by making it inaccessible. **XMEM_UNHIDE** Processes the page after DMA I/O. Also, this flag reveals the page and makes it accessible for cache-inconsistent platforms. **XMEM_WRITE_ONLY** Marks the intended transfer as outbound only. This flag is used with **XMEM_ACC_CHK** to indicate that read-only access to the page is sufficient.

Description

The **xmemdma** kernel service operates on the page specified by the *xaddr* parameter in the region specified by the cross-memory descriptor. If the cross-memory descriptor is for the kernel, the *xaddr* parameter specifies a kernel address. Otherwise, the *xaddr* parameter specifies the offset in the region described in the cross-memory descriptor.

The **xmemdma** kernel service is provided for machines that have processor-memory caches, but that do not perform DMA I/O through the cache. Device handlers for Micro Channel DMA devices use the **d_master** service and **d_complete** kernel service instead of the **xmemdma** kernel service.

If the *flag* parameter indicates **XMEM_HIDE** (that is, **XMEM_UNHIDE** is not set) and this is the first hide for the page, the **xmemdma** kernel service prepares the page for DMA I/O by flushing the cache and making the page invalid. When the **XMEM_UNHIDE** bit is set and this is the last unhide for the page, the following events take place:

1. The page is made valid.

   If the page is not in pager I/O state:
2. Any processes waiting on the page are readied.
3. The modified bit for the page is set unless the page has a read-only storage key.

The page is made not valid during DMA operations so that it is not addressable with any virtual address. This prevents any process from reading or loading any part of the page into the cache during the DMA operation.

The page specified must be in memory and must be pinned.

If the **XMEM_ACC_CHK** bit is set, then the **xmemdma** kernel service also verifies access permissions to the page. If the page access is read-only, then the **XMEM_WRITE_ONLY** bit must be set in the *flag* parameter.

Note:

1. The **xmemdma** kernel service does not hide or reveal the page nor does it perform any cache flushing. The service's primary function is for real-address translation.
2. This service is not supported for large-memory systems with greater than 4GB of physical memory addresses. For such systems, **xmemdma64** should be used.

Execution Environment

The **xmemdma** kernel service can be called from either the process or interrupt environment.

Return Values

On successful completion, the **xmemdma** service returns the real address corresponding to the *xaddr* and *xp* parameters.

Error Codes

The **xmemdma** kernel service returns a value of **XMEM_FAIL** if one of the following are true:

- The descriptor was invalid.
- The page specified by the *xaddr* or *xp* parameter is invalid.
- Access is not allowed to the page.

Related information:

Cross Memory Kernel Services

Understanding Direct Memory Access (DMA) Transfer

Dynamic Logical Partitioning

xmemdma64 Kernel Service

Purpose

Prepares a page for direct memory access (DMA) I/O or processes a page after DMA I/O is complete.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmem.h>
```

```
unsigned long long xmemdma64 (
struct xmem *dp,
caddr_t xaddr,>
int flags)
```

Parameters

Item	Description
dp	Specifies a cross-memory descriptor.
xaddr	Identifies the address that specify the page for transfer.
flags	Specifies whether to prepare a page for DMA I/O or process it after DMA I/O is complete. Possible values are: **XMEM_HIDE** Prepares the page for DMA I/O. If cache-inconsistent, then the data cache is flushed, the memory page is hidden, and the real page address is returned. If cache-consistent, then the modified bit is set and the real address of the page is returned. **XMEM_UNHIDE** Processes the page after DMA I/O. Also, this flag reveals the page, readies any waiting processes on the page, and sets the modified bit accordingly. **XMEM_ACC_CHK** Performs access checking on the page. When this flag is set, the page protection attributes are verified. **XMEM_WRITE_ONLY** Marks the intended transfer as outbound only. This flag is used with **XMEM_ACC_CHK** to indicate that read-only access to the page is sufficient.

Description

The **xmemdma64** kernel service operates on the page that is specified by the *xaddr* parameter in the region that is 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 that is described in the cross-memory descriptor.

The **xmemdma64** kernel service is provided for machines that have processor-memory caches, but that do not perform DMA I/O through the cache.

If the *flag* parameter indicates **XMEM_HIDE** (that is, **XMEM_UNHIDE** is not set) and it is the first hide for the page, the **xmemdma64** kernel service prepares the page for DMA I/O by flushing the cache and making the page invalid. When the **XMEM_UNHIDE** bit is set and it is the last unhide for the page, the following events take place:

1. The page is made valid.

   If the page is not in pager I/O state:
2. Any processes that is 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. It prevents any process from reading or loading any part of the page into the cache during the DMA operation.

The page that is specified must be in memory and must be pinned.

If the **XMEM_ACC_CHK** bit is set, then the **xmemdma64** kernel service also verifies access permissions to the page. If the page access is read-only, then the **XMEM_WRITE_ONLY** bit must be set in the *flag* parameter.

Note: The **xmemdma64** kernel service does not hide or reveal the page, nor does it perform any cache flushing. The service's primary function is for real-address translation.

Execution Environment

The **xmemdma64** kernel service can be called from either the process or interrupt environment.

Return Values

On successful completion, the **xmemdma64** service returns the real address corresponding to the *xaddr* and *xp* parameters.

Error Codes

The **xmemdma64** kernel service returns a value of **XMEM_FAIL** if one of the following are true:

- The descriptor was invalid.
- The page that is specified by the *xaddr* or *xp* parameter is invalid.
- Access is not allowed to the page.

Related information:

Cross Memory Kernel Services

Understanding Direct Memory Access (DMA) Transfer

xmempin Kernel Service

Purpose

Pins the specified address range in user or system memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int xmempin( base,  len,  xd)
caddr_t base;
int len;
struct xmem *xd;
```

Parameters

Item	Description
base	Specifies the address of the first byte to pin.
len	Indicates the number of bytes to pin.
xd	Specifies the cross-memory descriptor.

Description

The **xmempin** kernel service is used to pin pages backing a specified memory region which is defined in either system or user address space. Pinning a memory region prohibits the pager from stealing pages from the pages backing the pinned memory region. Once a memory region is pinned, accessing that region does not result in a page fault until the region is subsequently unpinned.

The cross-memory descriptor must have been filled in correctly prior to the **xmempin** call (for example, by calling the **xmattach** kernel service).

Execution Environment

The **xmempin** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates successful completion.
EFAULT	Indicates that the memory region as specified by the *base* and *len* parameters is not within the address space specified by the *xd* parameter.
EINVAL	Indicates that the value of the length parameter is negative or 0. Otherwise, the area of memory beginning at the byte specified by the *base* parameter and extending for the number of bytes specified by the *len* parameter is not defined.
ENOMEM	Indicates that the **xmempin** kernel service is unable to pin the region due to insufficient real memory or because it has exceeded the systemwide pin count.

Related reference:

"pin Kernel Service" on page 407

"xmemunpin Kernel Service"

Related information:

Memory Kernel Services

xmemunpin Kernel Service

Purpose

Unpins the specified address range in user or system memory.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>

int xmemunpin ( base,  len,  xd)
caddr_t base;
int len;
struct xmem *xd;
```

Parameters

Item	Description
base	Specifies the address of the first byte to unpin.
len	Indicates the number of bytes to unpin.
xd	Specifies the cross-memory descriptor.

Description

The **xmemunpin** kernel service unpins a region of memory. When the pin count is 0, the page is not pinned and can be paged out of real memory. Upon finding an unpinned page, the **xmemunpin** kernel service returns the **EINVAL** error code and leaves any remaining pinned pages still pinned.

The **xmemunpin** service should be used where the address space might be in either user or kernel space.

The cross-memory descriptor must have been filled in correctly prior to the **xmempin** call (for example, by calling the **xmattach** kernel service).

Execution Environment

The **xmemunpin** kernel service can be called in the process environment when unpinning data that is in either user space or system space. It can be called in the interrupt environment only when unpinning data that is in system space.

Return Values

Item	Description
0	Indicates successful completion.
EFAULT	Indicates that the memory region as specified by the *base* and *len* parameters is not within the address specified by the *xd* parameter.
EINVAL	Indicates that the value of the length parameter is negative or 0. Otherwise, the area of memory beginning at the byte specified by the *base* parameter and extending for the number of bytes specified by the *len* parameter is not defined. If neither cause is responsible, an unpinned page was specified.

Related reference:

"unpin Kernel Service" on page 519

"xmempin Kernel Service" on page 604

Related information:

Understanding Execution Environments

xmemzero Kernel Service

Purpose

Zeros a buffer described by a cross memory descriptor.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vmuser.h>
```

int xmemzero (*dp*, *uaddr*, *count***)**
struct xmem * *dp***;**
caddr_t *uaddr***;**
long *count***;**

Parameters

Item	Description
dp	The cross memory descriptor.
uaddr	The address in the buffer to begin zeroing.
count	The number of bytes to be zeroed.

Description

The **xmemzero** kernel service zeros a buffer described by a cross memory descriptor. The page specified must be in memory.

Execution Environment

The **xmemzero** kernel service can be called from a process or an interrupt environment.

Return Values

Item	Description
XMEM_SUCC	Indicates the area in the buffer has been zeroed.
XMEM_FAIL	Indicates one of the following errors: • The descriptor is marked by **XMEM_REMIO**. • The descriptor is not marked by **XMEM_PROC** and **XMEM_GLOBAL**. • Count < 0.

Related information:

Memory Kernel Services

Understanding Virtual Memory Manager Interfaces

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

Item	Description
uaddr	Specifies the address in memory specified by a cross-memory descriptor.
kaddr	Specifies the address in kernel memory.
count	Specifies the number of bytes to copy.
dp	Specifies the cross-memory descriptor.

Description

The **xmemin** kernel service performs a cross-memory move. A cross-memory move occurs when data is moved to or from an address space other than the address space that the program is executing in. The **xmemin** kernel service copies data from the specified address space to kernel global memory.

The **xmemin** kernel service is provided so that kernel processes and interrupt handlers can safely access a buffer within a user process. Calling the **xmattach** kernel service prepares the user buffer for the cross-memory move.

The **xmemin** kernel service differs from the **copyin** and **copyout** kernel services in that it is used to access a user buffer when not executing under the user process. In contrast, the **copyin** and **copyout** kernel services are used only to access a user buffer while executing under the user process.

Execution Environment

The **xmemin** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
XMEM_SUCC	Indicates successful completion.
XMEM_FAIL	Indicates one of the following errors: • The user does not have the appropriate access authority for the user buffer. • The user buffer is located in an address range that is not valid. • The segment containing the user buffer has been deleted. • The cross-memory descriptor is not valid. • A paging I/O error occurred while the user buffer was being accessed. If the user buffer is not in memory, the **xmemin** kernel service also returns an **XMEM_FAIL** error when executing on an interrupt level.

Related reference:

"xmattach Kernel Service" on page 598

"xmemout Kernel Service"

Related information:

Cross Memory Kernel Services

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

Item	Description
kaddr	Specifies the address in kernel memory.
uaddr	Specifies the address in memory specified by a cross-memory descriptor.
count	Specifies the number of bytes to copy.
dp	Specifies the cross-memory descriptor.

Description

The **xmemout** kernel service performs a cross-memory move. A cross-memory move occurs when data is moved to or from an address space other than the address space that the program is executing in. The **xmemout** kernel service copies data from kernel global memory to the specified address space.

The **xmemout** kernel service is provided so that kernel processes and interrupt handlers can safely access a buffer within a user process. Calling the **xmattach** kernel service prepares the user buffer for the cross-memory move.

The **xmemout** kernel service differs from the **copyin** and **copyout** kernel services in that it is used to access a user buffer when not executing under the user process. In contrast, the **copyin** and **copyout** kernel services are only used to access a user buffer while executing under the user process.

Execution Environment

The **xmemout** kernel service can be called from either the process or interrupt environment.

Return Values

Item	Description
XMEM_SUCC	Indicates successful completion.
XMEM_FAIL	Indicates one of the following errors: • The user does not have the appropriate access authority for the user buffer. • The user buffer is located in an address range that is not valid. • The segment containing the user buffer has been deleted. • The cross-memory descriptor is not valid. • A paging I/O error occurred while the user buffer was being accessed. If the user buffer is not in memory, the **xmemout** service also returns an **XMEM_FAIL** error when executing on an interrupt level.

Related reference:
"xmattach Kernel Service" on page 598
"xmemin Kernel Service" on page 607
Related information:
Cross Memory Kernel Services

xmempsize Kernel Service

Purpose

Reports the page size being used for a specified address range on the 64-bit kernel.

Syntax

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/xmem.h>

long long xmempsize (dp, uaddr, count)

struct xmem * dp;
void * uaddr;
size_t count;
```

Description

The **xmempsize** kernel service returns the size, in bytes, of the virtual memory pages contained in the memory range starting at *uaddr* and continuing for *count* number of bytes. If the memory range consists of virtual memory pages of different sizes, the size of the smallest pages contained in the range is returned.

The cross-memory descriptor, *dp*, must have been previously initialized to describe the buffer containing the specified range of memory. The **xmattach()** kernel service prepares a buffer and cross-memory descriptor for use with the **xmempsize()** kernel service.

Parameters

Item	Description
dp	Specifies the cross-memory descriptor.
uaddr	Specifies the starting address of the memory range.
count	Specifies the number of bytes.

Execution Environment

The **xmempsize** kernel service can be called from either the process or interrupt environment.

The **xmempsize** kernel service is only supported on the 64-bit kernel.

Return Values

On successful completion, the **xmempsize()** kernel service returns a page size in bytes.

Otherwise, the **xmempsize()** kernel service returns **XMEM_FAIL**.

Related reference:

"xmattach Kernel Service" on page 598

Related information:

Cross Memory Kernel Services

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

Item	Description
ptr	Specifies the address of the area in memory to free.
heap	Specifies the address of the heap from which the memory was allocated.

Description

The **xmfree** kernel service frees the area of memory pointed to by the *ptr* parameter in the heap specified by the *heap* parameter. This area of memory must be allocated with the **xmalloc** kernel service. In addition, the *ptr* pointer must be the pointer returned from the corresponding **xmalloc** call.

For example, the **xmfree** (*ptr*, **kernel_heap**) kernel service frees the area in the kernel heap allocated by *ptr*=**xmalloc** (*size*, *align*, **kernel_heap**).

A kernel extension must explicitly free any memory it allocates. If it does not, eventually subsequent allocations are unsuccessful. Pinned memory must also be unpinned before it is freed if allocated from the **kernel_heap**. The kernel does not keep track of which kernel extension owns various allocated areas in the heap. Therefore, the kernel never automatically frees these allocated areas on process termination or device close.

An additional interface to the **xmfree** kernel service is provided. The **free** (*ptr*) is equivalent to **xmfree** (*ptr*, **kernel_heap**).

Execution Environment

The **xmfree** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Indicates successful completion.
-1	Indicates one of the following errors: • The area to be freed was not allocated with the **xmalloc** kernel service. • The heap was not initialized for memory allocation.

Related reference:

"xmalloc Kernel Service" on page 597

Related information:

Memory Kernel Services

xmgethkeyset Kernel Service

Purpose

Retrieves the hardware keyset associated with a cross-memory descriptor.

Syntax

```
#include <sys/types.h>
#include <sys/kerrno.h>
#include <sys/xmem.h>
#include <sys/skeys.h>

kerrno_t xmgethkeyset (dp, keyset, flags)
struct xmem * dp;
hkeyset_t * hkeyset;
long flags;
```

Parameters

Item	Description
dp	Specifies a valid cross-memory descriptor.
hkeyset	Pointer to returned hardware keyset associated with the cross-memory descriptor.
flags	Must be set to zero.

Description

The **xmgethkeyset()** kernel service can be used to obtain the keyset associated with a cross-memory descriptor.

Kernel-key protection can be enforced on memory regions described by a cross-memory descriptor. The enforcement is done during normal access checking performed by cross-memory services, such as **xmemdma()** service.

Execution Environment

The **xmgethkeyset** kernel service can be called from the process or interrupt environment.

Return Values

Item	Description
0	Successful.
EINVAL_XMGETHKEYSET	Invalid parameter.

Related reference:

"xmsethkeyset Kernel Service"

"xmattach Kernel Service" on page 598

xmsethkeyset Kernel Service

Purpose

Alters hardware keyset associated with a cross-memory descriptor.

Syntax

```
#include <sys/types.h>
#include <sys/kerrno.h>
#include <sys/xmem.h>
#include <sys/skeys.h>

kerrno_t xmsethkeyset (dp, hkeyset, flags)
struct xmem * dp;
hkeyset_t hkeyset;
long flags;
```

Parameters

Item	Description
dp	Specifies a valid cross-memory descriptor.
hkeyset	Hardware keyset to assign to the cross-memory descriptor.
flags	Must be set to zero.

Description

The **xmsethkeyset()** kernel service can be used to modify the keyset associated with a cross-memory descriptor.

Kernel-key protection can be enforced on memory regions described by a cross-memory descriptor. The enforcement is done during normal access checking performed by cross-memory services, such as the **xmemdma()** service.

Execution Environment

The **xmsethkeyset** kernel service can be called from the process environment only.

Return Values

Item	Description
0	Successful.
EINVAL_XMSETHKEYSET	Invalid parameter or execution environment.

Related reference:

"xmgethkeyset Kernel Service" on page 611

"xmattach Kernel Service" on page 598

Device Driver Operations

This topic provides a description of standard device driver entry points parameters.

Standard Parameters to Device Driver Entry Points

Purpose

Provides a description of standard device driver entry points parameters.

Description

There are three parameters passed to device driver entry points that always have the same meanings: the *devno* parameter, the *chan* parameter, and the *ext* parameter.

The devno Parameter

This value, defined to be of type **dev_t**, specifies the device or subdevice to which the operation is directed. For convenience and portability, the **/usr/include/sys/sysmacros.h** file defines the following macros for manipulating device numbers:

Macro	Descriptionf
major(*devno***)**	Returns the major device number.
minor(*devno***)**	Returns the minor device number.
makedev(*maj*, *min***)**.	Constructs a composite device number in the format of *devno* from the major and minor device numbers given.

The chan Parameter

This value, defined to be of type **chan_t**, is the channel ID for a multiplexed device driver. If the device driver is not multiplexed, *chan* has the value of 0. If the driver is multiplexed, then the *chan* parameter is the **chan_t** value returned from the device driver's **ddmpx** routine.

The ext Parameter

The *ext* parameter, or extension parameter, is defined to be of type **int**. It is meaningful only with calls to such extended subroutines as the **openx**, **readx**, **writex**, and **ioctlx** subroutines. These subroutines allow applications to pass an extra, device-specific parameter to the device driver. This parameter is then passed to the **ddopen**, **ddread**, **ddwrite**, and **ddioctl** device driver entry points as the *ext* parameter. If the application uses one of the non-extended subroutines (for example, the **read** instead of the **readx** subroutine), then the *ext* parameter has a value of 0.

Note: Using the *ext* parameter is highly discouraged because doing so makes an application program less portable to other operating systems.

Related reference:

"ddioctl Device Driver Entry Point" on page 625

Related information:

read subroutine

Device Driver Kernel Extension Overview

buf Structure

Purpose

Describes buffering data transfers between a program and the peripheral device

Introduction to Kernel Buffers

For block devices, kernel buffers are used to buffer data transfers between a program and the peripheral device. These buffers are allocated in blocks of 4096 bytes. At any given time, each memory block is a member of one of two linked lists that the device driver and the kernel maintain:

List	**Description**
Available buffer queue (avlist)	A list of all buffers available for use. These buffers do not contain data waiting to be transferred to or from a device.
Busy buffer queue (blist)	A list of all buffers that contain data waiting to be transferred to or from a device.

Each buffer has an associated buffer header called the **buf** structure pointing to it. Each buffer header has several parts:

- Information about the block
- Flags to show status information
- Busy list forward and backward pointers
- Available list forward and backward pointers

The device driver maintains the `av_forw` and `av_back` pointers (for the available blocks), while the kernel maintains the `b_forw` and `b_back` pointers (for the busy blocks).

buf Structure Variables for Block I/O

The **buf** structure, which is defined in the **/usr/include/sys/buf.h** file, includes the following fields:

Item **Description**

`b_flags` Flag bits. The value of this field is constructed by logically ORing 0 or more of the following values:

B_WRITE
This operation is a write operation.

B_READ
This operation is a read data operation, rather than write.

B_DONE
I/O on the buffer has been done, so the buffer information is more current than other versions.

B_ERROR
A transfer error has occurred and the transaction has aborted.

B_BUSY
The block is not on the free list.

B_INFLIGHT
This I/O request has been sent to the physical device driver for processing.

B_AGE The data is not likely to be reused soon, so prefer this buffer for reuse. This flag suggests that the buffer goes at the head of the free list rather than at the end.

B_ASYNC
Asynchronous I/O is being performed on this block. When I/O is done, release the block.

B_DELWRI
The contents of this buffer still need to be written out before the buffer can be reused, even though this block may be on the free list. This is used by the **write** subroutine when the system expects another write to the same block to occur soon.

B_NOHIDE
Indicates that the data page should not be hidden during direct memory access (DMA) transfer.

B_SETMOD
Allows an enhanced I/O file system to cause a page to be considered modified.

B_STALE
The data conflicts with the data on disk because of an I/O error.

B_ XREADONLY
Indicates a read-only page in the external pager buffer list.

B_MORE_DONE
When set, indicates to the receiver of this **buf** structure that more structures are queued in the **IODONE** level. This permits device drivers to handle all completed requests before processing any new requests.

B_SPLIT
When set, indicates that the transfer can begin anywhere within the data buffer.

`b_forw` The forward busy block pointer.

`b_back` The backward busy block pointer.

`av_forw` The forward pointer for a driver request queue.

`av_back` The backward pointer for a driver request queue.

`b_iodone` Anyone calling the strategy routine must set this field to point to their I/O done routine. This routine is called on the **INTIODONE** interrupt level when I/O is complete.

`b_dev` The major and minor device number.

`b_bcount` The byte count for the data transfer.

`b_un.b_addr` The memory address of the data buffer.

`b_blkno` The block number on the device.

`b_resid` Amount of data not transferred after error.

`b_event` Anchor for event list.

`b_xmemd` Cross-memory descriptor.

Related reference:

"ddstrategy Device Driver Entry Point" on page 634
"bufx Structure"
Related information:
write subroutine
Device Driver Kernel Extension Overview

bufx Structure

Purpose

Extends the **buf** structure to accommodate new fields as needed for performance and RAS reasons.

Description

The **bufx** structure is available for use by the 64-bit kernel and 64-bit kernel extensions. The 32-bit kernel and 32-bit kernel extensions only have the option of using the **buf** structure.

bufx Structure Variables for Block I/O

The **bufx** structure, which is defined in the **/usr/include/sys/buf.h** file, includes the following fields:

Item **Description**

b_flags
Flag bits. The value of this field is constructed by the logical OR operation with 0 or more of the following values:

B_WRITE
This operation is a write operation.

B_READ
This operation is a read data operation.

B_DONE
I/O on the buffer is done, so the buffer information is more current than other versions.

B_ERROR
A transfer error occurred and the transaction aborted.

B_BUSY
The block is not on the free list.

B_INFLIGHT
This I/O request was sent to the physical device driver for processing.

B_AGE The data is not likely to be reused soon, so prefer this buffer for reuse. This flag suggests that the buffer goes at the head of the free list rather than at the end.

B_ASYNC
Asynchronous I/O is being performed on this block. When I/O is done, release the block.

B_DELWRI
The contents of this buffer still need to be written out before the buffer can be reused, even though this block may be on the free list. This is used by the **write** subroutine when the system expects another write to the same block to occur soon.

B_NOHIDE
Indicates that the data page should not be hidden during direct memory access (DMA) transfer.

B_STALE
The data conflicts with the data on disk because of an I/O error.

B_MORE_DONE
When set, indicates to the receiver of this **bufx** structure that more structures are queued in the **IODONE** level. This permits device drivers to handle all completed requests before processing any new requests.

B_SPLIT
When set, indicates that the transfer can begin anywhere within the data buffer.

B_BUFX A buffer is identified as an extended **buf** structure if all of the following conditions are met:

- **B_BUFX** bit is set in the b_flags field.
- The pointer obtained by recombining the bx_refptrtop field and the bx_refptrbot field points to the beginning of the structure.
- The bx_eyecatcher field, which identifies whether the **buf** structure is extended or not, is equal to the ASCII string "bufx".

B_BUFX_INITIAL
When set, indicates that the **buf** is extended.

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.

Item	Description
b_resid	The amount of data not transferred after error.
b_event	The anchor for event list.
b_xmemd	The cross-memory descriptor.
bx_refptrtop	The top half of the reference pointer.
bx_refptrbot	The bottom half of the reference pointer.
bx_version	The version of the **bufx** structure.
bx_eyecatcher	The field contains the string "bufx", allowing for easy identification of the **bufx** structure in KDB when dumping data and for structure verification in addition to using the **BUFX_VALIDATE** macro.
bx_flags	**Bufx** flags with a 64-bit field that can be used for **bufx**-specific flags that are yet to be defined.
bx_io_priority	If the underlying storage devices do not support I/O priority, this value is ignored. The bx_io_priority must be either the value of IOPRIORITY_UNSET (0) or a value from 1 to 15. Lower I/O priority values are considered to be more important than higher values. For example, a value of 1 is considered the highest priority and a value of 15 is considered the lowest priority. The value of IOPRIORITY_UNSET is defined in the **sys/extendio.h** file.
bx_io_cache_hint	If the underlying storage devices do not support I/O cache hints, this value is ignored. The bx_io_cache_hint must be either the value of CH_AGE_OUT_FAST or the value of CH_PAGE_WRITE (defined in the **sys/extendio.h** file). These values are mutually exclusive. If CH_AGE_OUT_FAST is set, the I/O buffer can be aged out quickly from the storage device buffer cache. This is useful in the situations where the application is already caching the I/O buffer and redundant caching within the storage layer can be avoided. If CH_PAGE_WRITE is set, the I/O buffer is written only to the storage device cache and not to the disk.

Related reference:

"buf Structure" on page 614

Character Lists Structure

Character device drivers, and other character-oriented support that can perform character-at-a-time I/O, can be implemented by using a common set of services and data buffers to handle characters in the form of *character lists*. A *character list* is a list or queue of characters. Some routines put characters in a list, and others remove the characters from the list.

Character lists, known as **clists**, contain a **clist** header and a chain of one or more data buffers known as character blocks**.** Putting characters on a queue allocates space (character blocks) from the common pool and links the character block into the data structure defining the character queue. Obtaining characters from a queue returns the corresponding space back to the pool.

A character list can be used to communicate between a character device driver top and bottom half. The **clist** header and the character blocks that are used by these routines must be pinned in memory, since they are accessed in the interrupt environment.

Users of the character list services must register (typically in the device driver **ddopen** routine) the number of character blocks to be used at any one time. This allows the kernel to manage the number of pinned character blocks in the character block pool. Similarly, when usage terminates (for example, when the device driver is closed), the using routine should remove its registration of character blocks. The **pincf** kernel service provides registration for character block usage.

The kernel provides four services for obtaining characters or character blocks from a character list: the **getc**, **getcb**, **getcbp**, and **getcx** kernel services. There are also four services that add characters or character blocks to character lists: the **putc**, **putcb**, **putcbp**, and **putcx** kernel services. The **getcf** kernel services allocates a free character block while the **putcf** kernel service returns a character block to the free list. Additionally, the **putcfl** kernel service returns a list of character buffers to the free list. The **waitcfree** kernel service determines if any character blocks are on the free list, and waits for one if none are available.

Using a Character List

For each character list you use, you must allocate a **clist** header structure. This **clist** structure is defined in the **/usr/include/sys/cblock.h** file.

You do not need to be concerned with maintaining the fields in the **clist** header, as the character list services do this for you. However, you should initialize the `c_cc` count field to 0, and both character block pointers (`c_cf` and `c_cl`) to null before using the **clist** header for the first time. The **clist** structure defines these fields.

Each buffer in the character list is a **cblock** structure, which is also defined in the **/usr/include/sys/cblock.h** file.

A character block data area does not need to be completely filled with characters. The **c_first** and **c_last** fields are zero-based offsets within the **c_data** array, which actually contains the data.

Only a limited amount of memory is available for character buffers. All character drivers share this pool of buffers. Therefore, you must limit the number of characters in your character list to a few hundred. When the device is closed, the device driver should make certain all of its character lists are flushed so the buffers are returned to the list of free buffers.

Related reference:

"getc Kernel Service" on page 181

"putc Kernel Service" on page 422

Related information:

Device Driver Kernel Extension Overview

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

Item	Description
devno	Specifies the major and minor device numbers of the device instance to close.
chan	Specifies the channel number.

Description

The **ddclose** entry point is called when a previously opened device instance is closed by the **close** subroutine or **fp_close** kernel service. The kernel calls the routine under different circumstances for non-multiplexed and multiplexed device drivers.

For non-multiplexed device drivers, the kernel calls the **ddclose** routine when the last process having the device instance open closes it. This causes the g-node reference count to be decremented to 0 and the g-node to be deallocated.

For multiplexed device drivers, the **ddclose** routine is called for each close associated with an explicit open. In other words, the device driver's **ddclose** routine is invoked once for each time its **ddopen** routine was invoked for the channel.

In some instances, data buffers should be written to the device before returning from the **ddclose** routine. These are buffers containing data to be written to the device that have been queued by the device driver but not yet written.

Non-multiplexed device drivers should reset the associated device to an idle state and change the device driver device state to closed. This can involve calling the **fp_close** kernel service to issue a close to an associated open device handler for the device. Returning the device to an idle state prevents the device from generating any more interrupt or direct memory access (DMA) requests. DMA channels and interrupt levels allocated for this device should be freed, until the device is re-opened, to release critical system resources that this device uses.

Multiplexed device drivers should provide the same device quiescing, but not in the **ddclose** routine. Returning the device to the idle state and freeing its resources should be delayed until the **ddmpx** routine is called to deallocate the last channel allocated on the device.

In all cases, the device instance is considered closed once the **ddclose** routine has returned to the caller, even if a nonzero return code is returned.

Execution Environment

The **ddclose** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddclose** entry point can indicate an error condition to the user-mode application program by returning a nonzero return code. This causes the subroutine call to return a value of -1. It also makes the return code available to the user-mode application in the **errno** global variable. The return code used should be one of the values defined in the **/usr/include/sys/errno.h** file.

The device is always considered closed even if a nonzero return code is returned.

When applicable, the return values defined in the POSIX 1003.1 standard for the **close** subroutine should be used.

Related reference:

"ddopen Device Driver Entry Point" on page 628

"fp_close Kernel Service" on page 144

Related information:

Programming in the Kernel Environment Overview

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

Item	Description
devno	Specifies the major and minor device numbers.
cmd	Specifies the function to be performed by the **ddconfig** routine.
uiop	Points to a **uio** structure describing the relevant data area for configuration information.

Description

The **ddconfig** entry point is used to configure a device driver. It can be called to do the following tasks:

- Initialize the device driver.
- Terminate the device driver.
- Request configuration data for the supported device.
- Perform other device-specific configuration functions.

The **ddconfig** routine is called by the device's Configure, Unconfigure, or Change method. Typically, it is called once for each device number (major and minor) to be supported. This is, however, device-dependent. The specific device method and **ddconfig** routine determines the number of times it is called.

The **ddconfig** routine can also provide additional device-specific functions relating to configuration, such as returning device vital product data (VPD). The **ddconfig** routine is usually invoked through the **sysconfig** subroutine by the device-specific Configure method.

Device drivers and their methods typically support these values for the *cmd* parameter:

Value **Description**

CFG_INIT Initializes the device driver and internal data areas. This typically involves the minor number specified by the *devno* parameter, for validity. The device driver's **ddconfig** routine also installs the device driver's entry points in the device switch table, if this was the first time called (for the specified major number). This can be accomplished by using the **devswadd** kernel service along with a **devsw** structure to add the device driver's entry points to the device switch table for the major device number supplied in the *devno* parameter.

The **CFG_INIT** command parameter should also copy the device-dependent information (found in the device-dependent structure provided by the caller) into a static or dynamically allocated save area for the specified device. This information should be used when the **ddopen** routine is later called.

The device-dependent structure's address and length are described in the **uio** structure pointed to by the *uiop* parameter. The **uiomove** kernel service can be used to copy the device-dependent structure into the device driver's data area.

When the **ddopen** routine is called, the device driver passes device-dependent information to the routines or other device drivers providing the device handler role in order to initialize the device. The delay in initializing the device until the **ddopen** call is received is useful in order to delay the use of valuable system resources (such as DMA channels and interrupt levels) until the device is actually needed.

Value **Description**

CFG_TERM Terminates the device driver associated with the specified device number, as represented by the *devno* parameter. The **ddconfig** routine determines if any opens are outstanding on the specified *devno* parameter. If none are, the **CFG_TERM** command processing marks the device as terminated, disallowing any subsequent opens to the device. All dynamically allocated data areas associated with the specified device number should be freed.

If this termination removes the last minor number supported by the device driver from use, the **devswdel** kernel service should be called to remove the device driver's entry points from the device switch table for the specified *devno* parameter.

If opens are outstanding on the specified device, the terminate operation is rejected with an appropriate error code returned. The Unconfigure method can subsequently unload the device driver if all uses of it have been terminated.

To determine if all the uses of the device driver have been terminated, a device method can make a **sysconfig** subroutine call. By using the **sysconfig SYS_QDVSW** operation, the device method can learn whether or not the device driver has removed itself from the device switch table.

CFG_QVPD Queries device-specific vital product data (VPD).

For this function, the calling routine sets up a **uio** structure pointed at by the *uiop* parameter to the **ddconfig** routine. This **uio** structure defines an area in the caller's storage in which the **ddconfig** routine is to write the VPD. The **uiomove** kernel service can be used to provide the data copy operation.

The data area pointed at by the *uiop* parameter has two different purposes, depending on the *cmd* function. If the **CFG_INIT** command has been requested, the **uiop** structure describes the location and length of the device-dependent data structure (DDS) from which to read the information. If the **CFG_QVPD** command has been requested, the **uiop** structure describes the area in which to write vital product data information. The content and format of this information is established by the specific device methods in conjunction with the device driver.

The **uiomove** kernel service can be used to facilitate copying information into or out of this data area. The format of the **uio** structure is defined in the **/usr/include/sys/uio.h** file and described further in the **uio** structure.

Execution Environment

The **ddconfig** routine and its operations are called in the process environment only.

Return Values

The **ddconfig** routine sets the return code to 0 if no errors are detected for the operation specified. If an error is to be returned to the caller, a nonzero return code should be provided. The return code used should be one of the values defined in the **/usr/include/sys/errno.h** file.

If this routine was invoked by a **sysconfig** subroutine call, the return code is passed to its caller (typically a device method). It is passed by presenting the error code in the **errno** global variable and providing a -1 return code to the subroutine.

Related reference:

"devswadd Kernel Service" on page 70

"uiomove Kernel Service" on page 516

Related information:

sysconfig subroutine

Device Driver Kernel Extension Overview

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

Item	Description
devno	Specifies the major and minor device numbers.
uiop	Points to the **uio** structure describing the data area or areas to be dumped.
cmd	The parameter from the kernel dump function that specifies the operation to be performed.
arg	The parameter from the caller that specifies the address of a parameter block associated with the kernel dump command.
chan	Specifies the channel number.
ext	Specifies the extension parameter.

Description

The kernel dump routine calls the **dddump** entry point to set up and send dump requests to the device. The **dddump** routine is optional for a device driver. It is required only when the device driver supports a device as a target for a possible kernel dump.

If this is the case, it is important that the system state change as little as possible when performing the dump. As a result, the **dddump** routine should use the minimal amount of services in writing the dump data to the device.

The *cmd* parameter can specify any of the following dump commands:

Dump Command	Description
DUMPINIT	Initialization a device in preparation for supporting a system dump. The specified device instance must have previously been opened. The *arg* parameter points to a **dumpio_stat** structure, defined in **/usr/include/sys/dump.h**. This is used for returning device-specific status in case of an error. The **dddump** routine should pin all code and data that the device driver uses to support dump writing. This is required to prevent a page fault when actually performing a write of the dump data. (Pinned code should include the **dddump** routine.) The **pin** or **pincode** kernel service can be used for this purpose.

Dump Command **Description**

DUMPQUERY
Determines the maximum and minimum number of bytes that can be transferred to the device in one **DUMPWRITE** command. For network dumps, the address of the write routine used in transferring dump data to the network dump device is also sent. The *uiop* parameter is not used and is null for this command. The *arg* parameter is a pointer to a **dmp_query** structure, as defined in the **/usr/include/sys/dump.h** file.

The **dmp_query** structure contains the following fields:

`min_tsize`
Minimum transfer size (in bytes).

`max_tsize`
Maximum transfer size (in bytes).

`dumpwrite`
Address of the write routine.

The **DUMPQUERY** command returns the data transfer size information in the **dmp_query** structure pointed to by the *arg* parameter. The kernel dump function then uses a buffer between the minimum and maximum transfer sizes (inclusively) when writing dump data.

If the buffer is not the size found in the `max_tsize` field, then its size must be a multiple of the value in the `min_tsize` field. The `min_tsize` field and the `max_tsize` field can specify the same value.

DUMPSTART
Suspends current device activity and provide whatever setup of the device is needed before receiving a **DUMPWRITE** command. The *arg* parameter points to a **dumpio_stat** structure, defined in **/usr/include/sys/dump.h**. This is used for returning device-specific status in case of an error.

DUMPWRITE
Writes dump data to the target device. The **uio** structure pointed to by the *uiop* parameter specifies the data area or areas to be written to the device and the starting device offset. The *arg* parameter points to a **dumpio_stat** structure, defined in **/usr/include/sys/dump.h**. This is used for returning device-specific status in case of an error. Code for the **DUMPWRITE** command should minimize its reliance on system services, process dispatching, and such interrupt services as the **INTIODONE** interrupt priority or device hardware interrupts.
Note: The **DUMPWRITE** command must never cause a page fault. This is ensured on the part of the caller, since the data areas to be dumped have been determined to be in memory. The device driver must ensure that all of its code, data and stack accesses are to pinned memory during its **DUMPINIT** command processing.

DUMPEND
Indicates that the kernel dump has been completed. Any cleanup of the device state should be done at this time.

DUMPTERM
Indicates that the specified device is no longer a selected dump target device. If no other devices supported by this **dddump** routine have a **DUMPINIT** command outstanding, the **DUMPTERM** code should unpin any resources pinned when it received the **DUMPINIT** command. (The **unpin** kernel service is available for unpinning memory.) The **DUMPTERM** command is received before the device is closed.

DUMPREAD
Receives the acknowledgment packet for previous **DUMPWRITE** operations to a communications device driver. If the device driver receives the acknowledgment within the specified time, it returns a 0 and the response data is returned to the kernel dump function in the *uiop* parameter. If the device driver does not receive the acknowledgment within the specified time, it returns a value of **ETIMEDOUT**.

The *arg* parameter contains a timeout value in milliseconds.

Execution Environment

The **DUMPINIT dddump** operation is called in the process environment only. The **DUMPQUERY**, **DUMPSTART**, **DUMPWRITE**, **DUMPEND**, and **DUMPTERM dddump** operations can be called in both the process environment and interrupt environment.

Return Values

The **dddump** entry point indicates an error condition to the caller by returning a nonzero return code.

Related reference:

"devdump Kernel Service" on page 68

"dmp_add Kernel Service" on page 91

Related information:

Device Driver Kernel Extension Overview

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;
void *arg;
ulong  devflag;
chan_t  chan;
int  ext;
```

Description

When a program issues an **ioctl** or **ioctlx** subroutine call, the kernel calls the **ddioctl** routine of the specified device driver. The **ddioctl** routine is responsible for performing whatever functions are requested. In addition, it must return whatever control information has been specified by the original caller of the **ioctl** subroutine. The *cmd* parameter contains the name of the operation to be performed.

Most ioctl operations depend on the specific device involved. However, all ioctl routines must respond to the following command:

Item	Description
IOCINFO	Returns a **devinfo** structure (defined in the **/usr/include/sys/devinfo.h** file) that describes the device. (Refer to the description of the special file for a particular device in the Application Programming Interface.) Only the first two fields of the data structure need to be returned if the remaining fields of the structure do not apply to the device.

The *devflag* parameter indicates one of several types of information. It can give conditions in which the device was opened. (These conditions can subsequently be changed by the **fcntl** subroutine call.) Alternatively, it can tell which of two ways the entry point was invoked:

- By the file system on behalf of a using application
- Directly by a kernel routine using the **fp_ioctl** kernel service

Thus flags in the *devflag* parameter have the following definitions, as defined in the **/usr/include/sys/device.h** file:

Item	Description
DKERNEL	Entry point called by kernel routine using the **fp_ioctl** service.
DREAD	Open for reading.
DWRITE	Open for writing.
DAPPEND	Open for appending.
DNDELAY	Device open in nonblocking mode.

Parameters

Item	Description
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.

Execution Environment

The **ddioctl** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddioctl** entry point can indicate an error condition to the user-mode application program by returning a nonzero return code. This causes the **ioctl** subroutine to return a value of -1 and makes the return code available to the user-mode application in the **errno** global variable. The error code used should be one of the values defined in the **/usr/include/sys/errno.h** file.

When applicable, the return values defined in the POSIX 1003.1 standard for the **ioctl** subroutine should be used.

Related reference:

"Standard Parameters to Device Driver Entry Points" on page 613

"fp_ioctl Kernel Service" on page 150

Related information:

fcntl subroutine

Device Driver Kernel Extension Overview

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

Item	Description
devno	Specifies the major and minor device numbers.
chanp	Specifies the channel ID, passed by reference.
channame	Points to the path name extension for the channel to be allocated.

Description

Only multiplexed character class device drivers can provide the **ddmpx** routine, and *every* multiplexed driver must do so. The **ddmpx** routine cannot be provided by block device drivers even when providing *raw* read/write access.

A multiplexed device driver is a character class device driver that supports the assignment of channels to provide finer access control to a device or virtual subdevice. This type of device driver has the capability to decode special channel-related information appended to the end of the path name of the device's special file. This path name extension is used to identify a logical or virtual subdevice or channel.

When an **open** or **creat** subroutine call is issued to a device instance supported by a multiplexed device driver, the kernel calls the device driver's **ddmpx** routine to allocate a channel.

The kernel calls the **ddmpx** routine when a channel is to be allocated or deallocated. Upon allocation, the kernel dynamically creates g-nodes (in-core i-nodes) for channels on a multiplexed device to allow the protection attributes to differ for various channels.

To allocate a channel, the **ddmpx** routine is called with a *channame* pointer to the path name extension. The path name extension starts after the first **/** (slash) character that follows the special file name in the path name. The **ddmpx** routine should perform the following actions:

- Parse this path name extension.
- Allocate the corresponding channel.
- Return the channel ID through the *chanp* parameter.

If no path name extension exists, the *channame* pointer points to a null character string. In this case, an available channel should be allocated and its channel ID returned through the *chanp* parameter.

If no error is returned from the **ddmpx** routine, the returned channel ID is used to determine if the channel was already allocated. If already allocated, the g-node for the associated channel has its reference count incremented. If the channel was not already allocated, a new g-node is created for the channel. In either case, the device driver's **ddopen** routine is called with the channel number assigned by the **ddmpx** routine. If a nonzero return code is returned by the **ddmpx** routine, the channel is assumed not to have been allocated, and the device driver's **ddopen** routine is not called.

If a close of a channel is requested so that the channel is no longer used (as determined by the channel's g-node reference count going to 0), the kernel calls the **ddmpx** routine. The **ddmpx** routine deallocates the channel after the **ddclose** routine was called to close the last use of the channel. If a nonzero return code is returned by the **ddclose** routine, the **ddmpx** routine is still called to deallocate the channel. The **ddclose** routine's return code is saved, to be returned to the caller. If the **ddclose** routine returned no error, but a nonzero return code was returned by the **ddmpx** routine, the channel is assumed to be deallocated, although the return code is returned to the caller.

To deallocate a channel, the **ddmpx** routine is called with a null *channame* pointer and the channel ID passed by reference in the *chanp* parameter. If the channel g-node reference count has gone to 0, the kernel calls the **ddmpx** routine to deallocate the channel after invoking the **ddclose** routine to close it. The **ddclose** routine should not itself deallocate the channel.

Execution Environment

The **ddmpx** routine is called in the process environment only.

Return Values

If the allocation or deallocation of a channel is successful, the **ddmpx** routine should return a return code of 0. If an error occurs on allocation or deallocation, this routine returns a nonzero value.

The return code should conform to the return codes described for the **open** and **close** subroutines in the POSIX 1003.1 standard, where applicable. Otherwise, the return code should be one defined in the **/usr/include/sys/errno.h** file.

Related reference:

"ddclose Device Driver Entry Point" on page 619

"ddopen Device Driver Entry Point"

Related information:

Device Driver Kernel Extension Overview

ddopen Device Driver Entry Point

Purpose

Prepares a device for reading, writing, or control functions.

Syntax

```
#include <sys/device.h>
int ddopen (devno, devflag, chan,  ext)
dev_t  devno;
ulong  devflag;
chan_t  chan;
int ext;
```

Parameters

Item	Description
devno	Indicates major and minor device numbers.
devflag	Specifies open file control flags.
chan	Specifies the channel number.
ext	Specifies the extension parameter.

Description

The kernel calls the **ddopen** routine of a device driver when a program issues an **open** or **creat** subroutine call. It can also be called when a system call, kernel process, or other device driver uses the **fp_opendev** or **fp_open** kernel service to use the device.

The **ddopen** routine must first ensure exclusive access to the device, if necessary. Many character devices, such as printers and plotters, should be opened by only one process at a time. The **ddopen** routine can enforce this by maintaining a static flag variable, which is set to 1 if the device is open and 0 if not.

Each time the **ddopen** routine is called, it checks the value of the flag. If the value is other than 0, the **ddopen** routine returns with a return code of **EBUSY** to indicate that the device is already open. Otherwise, the **ddopen** routine sets the flag and returns normally. The **ddclose** entry point later clears the flag when the device is closed.

Since most block devices can be used by several processes at once, a block driver should not try to enforce opening by a single user.

The **ddopen** routine must initialize the device if this is the first open that has occurred. Initialization involves the following steps:

1. The **ddopen** routine should allocate the required system resources to the device (such as DMA channels, interrupt levels, and priorities). It should, if necessary, register its device interrupt handler for the interrupt level required to support the target device. (The **i_init** and **d_init** kernel services are available for initializing these resources.)
2. If this device driver is providing the head role for a device and another device driver is providing the handler role, the **ddopen** routine should use the **fp_opendev** kernel service to open the device handler.

   Note: The **fp_opendev** kernel service requires a *devno* parameter to identify which device handler to open. This *devno* value, taken from the appropriate device dependent structure (DDS), should have been stored in a special save area when this device driver's **ddconfig** routine was called.

Flags Defined for the devflag Parameter

The *devflag* parameter has the following flags, as defined in the **/usr/include/sys/device.h** file:

Item	Description
DKERNEL	Entry point called by kernel routine using the **fp_opendev** or **fp_open** kernel service.
DREAD	Open for reading.
DWRITE	Open for writing.
DAPPEND	Open for appending.
DNDELAY	Device open in nonblocking mode.

Execution Environment

The **ddopen** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddopen** entry point can indicate an error condition to the user-mode application program by returning a nonzero return code. Returning a nonzero return code causes the **open** or **creat** subroutines to return a value of -1 and makes the return code available to the user-mode application in the **errno** global variable. The return code used should be one of the values defined in the **/usr/include/errno.h** file.

If a nonzero return code is returned by the **ddopen** routine, the open request is considered to have failed. No access to the device instance is available to the caller as a result. In addition, for nonmultiplexed drivers, if the failed open was the first open of the device instance, the kernel calls the driver's **ddclose** entry point to allow resources and device driver state to be cleaned up. If the driver was multiplexed, the kernel does not call the **ddclose** entry point on an open failure.

When applicable, the return values defined in the POSIX 1003.1 standard for the **open** subroutine should be used.

Related reference:

"ddclose Device Driver Entry Point" on page 619

Related information:

close subroutine

Programming in the Kernel Environment Overview

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

Item	Description
devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure describing the data area or areas in which to be written.
chan	Specifies the channel number.
ext	Specifies the extension parameter.

Description

When a program issues a **read** or **readx** subroutine call or when the **fp_rwuio** kernel service is used, the kernel calls the **ddread** entry point.

This entry point receives a pointer to a **uio** structure that provides variables used to specify the data transfer operation.

Character device drivers can use the **ureadc** and **uiomove** kernel services to transfer data into and out of the user buffer area during a **read** subroutine call. These services receive a pointer to the **uio** structure and update the fields in the structure by the number of bytes transferred. The only fields in the **uio** structure that cannot be modified by the data transfer are the `uio_fmode` and `uio_segflg` fields.

For most devices, the **ddread** routine sends the request to the device handler and then waits for it to finish. The waiting can be accomplished by calling the **e_sleep** kernel service. This service suspends the driver and the process that called it and permits other processes to run until a specified event occurs.

When the I/O operation completes, the device usually issues an interrupt, causing the device driver's interrupt handler to be called. The interrupt handler then calls the **e_wakeup** kernel service specifying the awaited event, thus allowing the **ddread** routine to resume.

The `uio_resid` field initially contains the total number of bytes to read from the device. If the device driver supports it, the `uio_offset` field indicates the byte offset on the device from which the read should start.

The `uio_offset` field is a 64 bit integer (`offset_t`); this allows the file system to send I/O requests to a device driver's read & write entry points which have logical offsets beyond 2 gigabytes. Device drivers must use care not to cause a loss of significance by assigning the offset to a 32 bit variable or using it in calculations that overflow a 32 bit variable.

If no error occurs, the `uio_resid` field should be 0 on return from the **ddread** routine to indicate that all requested bytes were read. If an error occurs, this field should contain the number of bytes remaining to be read when the error occurred.

If a read request starts at a valid device offset but extends past the end of the device's capabilities, no error should be returned. However, the `uio_resid` field should indicate the number of bytes not transferred. If the read starts at the end of the device's capabilities, no error should be returned. However, the `uio_resid` field should not be modified, indicating that no bytes were transferred. If the read starts past the end of the device's capabilities, an **ENXIO** return code should be returned, without modifying the `uio_resid` field.

When the **ddread** entry point is provided for raw I/O to a block device, this routine usually translates requests into block I/O requests using the **uphysio** kernel service.

Execution Environment

The **ddread** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddread** entry point can indicate an error condition to the caller by returning a nonzero return code. This causes the subroutine call to return a value of -1. It also makes the return code available to the user-mode program in the **errno** global variable. The error code used should be one of the values defined in the **/usr/include/sys/errno.h** file.

When applicable, the return values defined in the POSIX 1003.1 standard for the **read** subroutine should be used.

Related reference:

“ddwrite Device Driver Entry Point” on page 636

“Select/Poll Logic for ddwrite and ddread Routines” on page 638

Related information:

read, readx

Programming in the Kernel Environment Overview

ddrevoke Device Driver Entry Point

Purpose

Ensures that a secure path to a terminal is provided.

Syntax

```
#include <sys/device.h>
#include <sys/types.h>

int ddrevoke ( devno,  chan,  flag)
dev_t devno;
chan_t chan;
int flag;
```

Parameters

Item	Description
devno	Specifies the major and minor device numbers.
chan	Specifies the channel number. For a multiplexed device driver, a value of -1 in this parameter means access to all channels is to be revoked.
flag	Currently defined to have the value of 0. (Reserved for future extensions.)

Description

The **ddrevoke** entry point can be provided only by character class device drivers. It cannot be provided by block device drivers even when providing raw read/write access. A **ddrevoke** entry point is required only by device drivers supporting devices in the Trusted Computing Path to a terminal (for example, by the **/dev/ lft** and **/dev/tty** files for the low function terminal and teletype device drivers). The **ddrevoke** routine is called by the **frevoke** and **revoke** subroutines.

The **ddrevoke** routine revokes access to a specific device or channel (if the device driver is multiplexed). When called, the **ddrevoke** routine should terminate all processes waiting in the device driver while accessing the specified device or channel. It should terminate the processes by sending a SIGKILL signal to all processes currently waiting for a specified device or channel data transfer. The current process is not to be terminated.

If the device driver is multiplexed and the channel ID in the *chan* parameter has the value -1, all channels are to be revoked.

Execution Environment

The **ddrevoke** routine is called in the process environment only.

Return Values

The **ddrevoke** routine should return a value of 0 for successful completion, or a value from the **/usr/include/errno.h** file on error.

Files

Item	Description
/dev/lft	Specifies the path of the LFT special file.
/dev/tty	Specifies the path of the tty special file.

Related information:

frevoke subroutine

revoke subroutine

TTY Subsystem Overview

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

Item	Description
devno	Specifies the major and minor device numbers.
events	Specifies the events to be checked.
reventp	Returned events pointer. This parameter, passed by reference, is used by the **ddselect** routine to indicate which of the selected events are true at the time of the call. The returned events location pointed to by the *reventp* parameter is set to 0 before entering this routine.
chan	Specifies the channel number.

Description

The **ddselect** entry point is called when the **select** or **poll** subroutine is used, or when the **fp_select** kernel service is invoked. It determines whether a specified event or events have occurred on the device.

Only character class device drivers can provide the **ddselect** routine. It cannot be provided by block device drivers even when providing raw read/write access.

Requests for Information on Events

The *events* parameter represents possible events to check as flags (bits). There are three basic events defined for the **select** and **poll** subroutines, when applied to devices supporting select or poll operations:

Event	Description
POLLIN	Input is present on the device.
POLLOUT	The device is capable of output.
POLLPRI	An exceptional condition has occurred on the device.

A fourth event flag is used to indicate whether the **ddselect** routine should record this request for later notification of the event using the **selnotify** kernel service. This flag can be set in the *events* parameter if the device driver is not required to provide asynchronous notification of the requested events:

Event	Description
POLLSYNC	This request is a synchronous request only. The routine need not call the **selnotify** kernel service for this request even if the events later occur.

Additional event flags in the *events* parameter are left for device-specific events on the **poll** subroutine call.

Select Processing

If one or more events specified in the *events* parameter are true, the **ddselect** routine should indicate this by setting the corresponding bits in the *reventp* parameter. Note that the *reventp* returned events parameter is passed by reference.

If none of the requested events are true, then the **ddselect** routine sets the returned events parameter to 0. It is passed by reference through the *reventp* parameter. It also checks the **POLLSYNC** flag in the *events* parameter. If this flag is true, the **ddselect** routine should just return, since the event request was a synchronous request only.

However, if the **POLLSYNC** flag is false, the **ddselect** routine must notify the kernel when one or more of the specified events later happen. For this purpose, the routine should set separate internal flags for each event requested in the *events* parameter.

When any of these events become true, the device driver routine should use the **selnotify** service to notify the kernel. The corresponding internal flags should then be reset to prevent re-notification of the event.

Sometimes the device can be in a state in which a supported event or events can never be satisfied (such as when a communication line is not operational). In this case, the **ddselect** routine should simply set the corresponding *reventp* flags to 1. This prevents the **select** or **poll** subroutine from waiting indefinitely. As a result however, the caller will not in this case be able to distinguish between satisfied events and unsatisfiable ones. Only when a later request with an **NDELAY** option fails will the error be detected.

Note: Other device driver routines (such as the **ddread**, **ddwrite** routines) may require logic to support select or poll operations.

Execution Environment

The **ddselect** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddselect** routine should return with a return code of 0 if the select or poll operation requested is valid for the resource specified. Requested operations are not valid, however, if either of the following is true:

- The device driver does not support a requested event.
- The device is in a state in which poll and select operations are not accepted.

In these cases, the **ddselect** routine should return with a nonzero return code (typically **EINVAL**), and without setting the relevant *reventp* flags to 1. This causes the **poll** subroutine to return to the caller with the **POLLERR** flag set in the returned events parameter associated with this resource. The **select** subroutine indicates to the caller that all requested events are true for this resource.

When applicable, the return values defined in the POSIX 1003.1 standard for the **select** subroutine should be used.

Related reference:

"fp_select Kernel Service" on page 166

Related information:

select subroutine

Programming in the Kernel Environment Overview

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

Item	Description
bp	Points to a **buf** structure describing all information needed to perform the data transfer.

Description

When the kernel needs a block I/O transfer, it calls the **ddstrategy** strategy routine of the device driver for that device. The strategy routine schedules the I/O to the device. This typically requires the following actions:

- The request or requests must be added on the list of I/O requests that need to be processed by the device.
- If the request list was empty before the preceding additions, the device's start I/O routine must be called.

Required Processing

The **ddstrategy** routine can receive a single request with multiple **buf** structures. However, it is not required to process requests in any specific order.

The strategy routine can be passed a list of operations to perform. The `av_forw` field in the **buf** header describes this null-terminated list of **buf** headers. This list is not doubly linked: the `av_back` field is undefined.

Block device drivers must be able to perform multiple block transfers. If the device cannot do multiple block transfers, or can only do multiple block transfers under certain conditions, then the device driver must transfer the data with more than one device operation.

Kernel Buffers and Using the buf Structure

An area of memory is set aside within the kernel memory space for buffering data transfers between a program and the peripheral device. Each kernel buffer has a header, the **buf** structure, which contains all necessary information for performing the data transfer. The **ddstrategy** routine is responsible for updating fields in this header as part of the transfer.

The caller of the strategy routine should set the `b_iodone` field to point to the caller's I/O done routine. When an I/O operation is complete, the device driver calls the **iodone** kernel service, which then calls the I/O done routine specified in the `b_iodone` field. The **iodone** kernel service makes this call from the **INTIODONE** interrupt level.

The value of the `b_flags` field is constructed by logically ORing zero or more possible `b_flags` field flag values.

Attention:

- Do not modify any of the following fields of the **buf** structure passed to the **ddstrategy** entry point: the `b_forw`, `b_back`, `b_dev`, `b_un`, or `b_blkno` field. Modifying these fields can cause unpredictable and disastrous results.
- Do not modify any of the following fields of a **buf** structure acquired with the **geteblk** service: the `b_flags`, `b_forw`, `b_back`, `b_dev`, `b_count`, or `b_un` field. Modifying any of these fields can cause unpredictable and disastrous results.

Execution Environment

The **ddstrategy** routine must be coded to execute in an interrupt handler execution environment (device driver bottom half). That is, the routine should neither touch user storage, nor page fault, nor sleep.

Return Values

The **ddstrategy** routine, unlike other device driver routines, does not return a return code. Any error information is returned in the appropriate fields within the **buf** structure pointed to by the *bp* parameter.

When applicable, the return values defined in the POSIX 1003.1 standard for the **read** and **write** subroutines must be used.

Related reference:

"geteblk Kernel Service" on page 185

Related information:

read subroutine

write subroutine

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

Item	Description
devno	Specifies the major and minor device numbers.
uiop	Points to a **uio** structure describing the data area or areas from which to be written.
chan	Specifies the channel number.
ext	Specifies the extension parameter.

Description

When a program issues a **write** or **writex** subroutine call or when the **fp_rwuio** kernel service is used, the kernel calls the **ddwrite** entry point.

This entry point receives a pointer to a **uio** structure, which provides variables used to specify the data transfer operation.

Character device drivers can use the **uwritec** and **uiomove** kernel services to transfer data into and out of the user buffer area during a **write** subroutine call. These services are passed a pointer to the **uio** structure. They update the fields in the structure by the number of bytes transferred. The only fields in the **uio** structure that are not potentially modified by the data transfer are the `uio_fmode` and `uio_segflg` fields.

For most devices, the **ddwrite** routine queues the request to the device handler and then waits for it to finish. The waiting is typically accomplished by calling the **e_sleep** kernel service to wait for an event. The **e_sleep** kernel service suspends the driver and the process that called it and permits other processes to run.

When the I/O operation is completed, the device usually causes an interrupt, causing the device driver's interrupt handler to be called. The interrupt handler then calls the **e_wakeup** kernel service specifying the awaited event, thus allowing the **ddwrite** routine to resume.

The `uio_resid` field initially contains the total number of bytes to write to the device. If the device driver supports it, the `uio_offset` field indicates the byte offset on the device from where the write should start.

The `uio_offset` field is a 64 bit integer (`offset_t`); this allows the file system to send I/O requests to a device driver's read & write entry points which have logical offsets beyond 2 gigabytes. Device drivers must use care not to cause a loss of significance by assigning the offset to a 32 bit variable or using it in calculations that overflow a 32 bit variable.

If no error occurs, the `uio_resid` field should be 0 on return from the **ddwrite** routine to indicate that all requested bytes were written. If an error occurs, this field should contain the number of bytes remaining to be written when the error occurred.

If a write request starts at a valid device offset but extends past the end of the device's capabilities, no error should be returned. However, the `uio_resid` field should indicate the number of bytes not transferred. If the write starts at or past the end of the device's capabilities, no data should be transferred. An error code of **ENXIO** should be returned, and the `uio_resid` field should not be modified.

When the **ddwrite** entry point is provided for raw I/O to a block device, this routine usually uses the **uphysio** kernel service to translate requests into block I/O requests.

Execution Environment

The **ddwrite** routine is executed only in the process environment. It should provide the required serialization of its data structures by using the locking kernel services in conjunction with a private lock word defined in the driver.

Return Values

The **ddwrite** entry point can indicate an error condition to the caller by returning a nonzero return value. This causes the subroutine to return a value of -1. It also makes the return code available to the user-mode program in the **errno** global variable. The error code used should be one of the values defined in the **/usr/include/sys/errno.h** file.

When applicable, the return values defined in the POSIX 1003.1 standard for the **write** subroutine should be used.

Related Information

The **ddread** device driver entry point.

The **CIO_GET_FASTWRT** ddioctl.

The **e_sleep** kernel service, **e_wakeup** kernel service, **fp_rwuio** kernel service, **uiomove** kernel service, **uphysio** kernel service, **uwritec** kernel service.

The **uio** structure.

The **write** and **writex** subroutines.

Device Driver Kernel Extension Overview, Understanding Device Driver Roles, Understanding Interrupts, Understanding Locking in *Kernel Extensions and Device Support Programming Concepts*.

Related reference:

"ddread Device Driver Entry Point" on page 630

Related information:

CIO_GET_FASTWRT subroutine

Device Driver Kernel Extension Overview

Select/Poll Logic for ddwrite and ddread Routines

Description

The **ddread** and **ddwrite** entry points require logic to support the **select** and **poll** operations. Depending on how the device driver is written, the interrupt routine may also need to include this logic as well.

The select/poll logic is required wherever code checks on the occurrence of desired events. At each point where one of the selection criteria is found to be true, the device driver should check whether a notification is due for that selection. If so, it should call the **selnotify** kernel service to notify the kernel of the event.

The *devno*, *chan*, and *revents* parameters are passed to the **selnotify** kernel service to indicate which device and which events have become true.

Related reference:

"ddread Device Driver Entry Point" on page 630

Related information:

poll subroutine

Device Driver Kernel Extension Overview

uio Structure

Purpose

Describes a memory buffer to be used in a data transfer.

Introduction

The user I/O or **uio** structure is a data structure describing a memory buffer to be used in a data transfer. The **uio** structure is most commonly used in the read and write interfaces to device drivers supporting character or raw I/O. It is also useful in other instances in which an input or output buffer can exist in different kinds of address spaces, and in which the buffer is not contiguous in virtual memory.

The **uio** structure is defined in the **/usr/include/sys/uio.h** file.

Description

The **uio** structure describes a buffer that is not contiguous in virtual memory. It also indicates the address space in which the buffer is defined. When used in the character device read and write interface, it also contains the device open-mode flags, along with the device read/write offset.

The kernel provides services that access data using a **uio** structure. The **ureadc**, **uwritec**, **uiomove**, and **uphysio** kernel services all perform data transfers into or out of a data buffer described by a **uio** structure. The **ureadc** kernel service writes a character into the buffer described by the **uio** structure. The **uwritec** kernel service reads a character from the buffer. These two services have names opposite from what you would expect, since they are named for the user action initiating the operation. A read on the part of the user thus results in a device driver writing to the buffer, while a write results in a driver reading from the buffer.

The **uiomove** kernel service copies data to or from a buffer described by a **uio** structure from or to a buffer in the system address space. The **uphysio** kernel service is used primarily by block device drivers providing raw I/O support. The **uphysio** kernel service converts the character read or write request into a block read or write request and sends it to the **ddstrategy** routine.

The buffer described by the **uio** structure can consist of multiple noncontiguous areas of virtual memory of different lengths. This is achieved by describing the data buffer with an array of elements, each of which consists of a virtual memory address and a byte length. Each element is defined as an `iovec` element. The **uio** structure also contains a field specifying the total number of bytes in the data buffer described by the structure.

Another field in the **uio** structure describes the address space of the data buffer, which can either be system space, user space, or cross-memory space. If the address space is defined as cross memory, an additional array of cross-memory descriptors is specified in the **uio** structure to match the array of `iovec` elements.

The **uio** structure also contains a byte offset (`uio_offset`). This field is a 64 bit integer (`offset_t`); it allows the file system to send I/O requests to a device driver's read & write entry points which have logical offsets beyond 2 gigabytes. Device drivers must use care not to cause a loss of significance by assigning the offset to a 32 bit variable or using it in calculations that overflow a 32 bit variable.

The called routine (device driver) is permitted to modify fields in the **uio** and **iovec** structures as the data transfer progresses. The final `uio_resid` count is in fact used to determine how much data was transferred. Therefore this count must be decremented, with each operation, by the number of bytes actually copied.

The **uio** structure contains the following fields:

Field	**Description**
`uio_iov`	A pointer to an array of **iovec** structures describing the user buffer for the data transfer.
`uio_xmem`	A pointer to an array of **xmem** structures containing the cross-memory descriptors for the **iovec** array.
`uio_iovcnt`	The number of yet-to-be-processed **iovec** structures in the array pointed to by the `uio_iov` pointer. The count must be at least 1. If the count is greater than 1, then a *scatter-gather* of the data is to be performed into or out of the areas described by the **iovec** structures.
`uio_iovdcnt`	The number of already processed **iovec** structures in the **iovec** array.
`uio_offset`	The file offset established by a previous **lseek**, **llseek** subroutine call. Most character devices ignore this variable, but some, such as the **/dev/mem** pseudo-device, use and maintain it.
`uio_segflg`	A flag indicating the type of buffer being described by the **uio** structure. This flag typically describes whether the data area is in user or kernel space or is in cross-memory. Refer to the **/usr/include/sys/uio.h** file for a description of the possible values of this flag and their meanings.
`uio_fmode`	The value of the file mode that was specified on opening the file or modified by the **fcntl** subroutine. This flag describes the file control parameters. The **/usr/include/sys/fcntl.h** file contains specific values for this flag.
`uio_resid`	The byte count for the data transfer. It must not exceed the sum of all the `iov_len` values in the array of **iovec** structures. Initially, this field contains the total byte count, and when the operation completes, the value must be decremented by the actual number of bytes transferred.

The **iovec** structure contains the starting address and length of a contiguous data area to be used in a data transfer. The **iovec** structure is the element type in an array pointed to by the `uio_iov` field in the **uio** structure. This array can contain any number of **iovec** structures, each of which describes a single unit of contiguous storage. Taken together, these units represent the total area into which, or from which, data is to be transferred. The `uio_iovcnt` field gives the number of **iovec** structures in the array.

The **iovec** structure contains the following fields:

Field	Description
iov_base	A variable in the **iovec** structure containing the base address of the contiguous data area in the address space specified by the uio_segflag field. The length of the contiguous data area is specified by the iov_len field.
iov_len	A variable in the **iovec** structure containing the byte length of the data area starting at the address given in the **iov_base** variable.

Related reference:

"uiomove Kernel Service" on page 516

"uphysio Kernel Service" on page 523

"vnop_getxacl Entry Point" on page 663

Related information:

Device Driver Kernel Extension Overview

Virtual File System Operations

The following topic provides entry points specified by the virtual file system interface for performing operations on vfs structures.

The following entry points are specified by the virtual file system interface for performing operations on **vfs** structures:

Entry Point	Description
vfs_aclxcntl	Issues ACL related control operations for a file system.
vfs_cntl	Issues control operations for a file system.
vfs_init	Initializes a virtual file system.
vfs_mount	Mounts a virtual file system.
vfs_root	Finds the root v-node of a virtual file system.
vfs_statfs	Obtains virtual file system statistics.
vfs_sync	Forces file system updates to permanent storage.
vfs_umount	Unmounts a virtual file system.
vfs_vget	Gets the v-node corresponding to a file identifier.

The following entry points are specified by the Virtual File System interface for performing operations on v-node structures:

Entry Point	Description
vnop_access	Tests a user's permission to access a file.
vnop_close	Releases the resources associated with a v-node.
vnop_create	Creates and opens a new file.
vnop_create_attr	Creates and opens a new file with initial attributes.
vnop_fclear	Releases portions of a file (by zeroing bytes).
vnop_fid	Builds a file identifier for a v-node.
vnop_finfo	Returns pathconf information about a file or file system.
vnop_fsync	Flushes in-memory information and data to permanent storage.
vnop_fsync_range	Flushes in-memory information and data for a given range to permanent storage.
vnop_ftrunc	Decreases the size of a file.
vnop_getacl	Gets information about access control, by retrieving the access control list.
vnop_getattr	Gets the attributes of a file.
vnop_getxacl	Gets information about access control by retrieving the ACL. Provides an advanced interface when compared to **vnop_getacl**.
vnop_hold	Assures that a v-node is not destroyed, by incrementing the v-node's use count.
vnop_ioctl	Performs miscellaneous operations on devices.
vnop_link	Creates a new directory entry for a file.
vnop_lockctl	Sets, removes, and queries file locks.
vnop_lookup	Finds an object by name in a directory.
vnop_map	Associates a file with a memory segment.

Entry Point	Description
vnop_map_lloff	Associates a file with a memory segment using 64 bit offset.
vnop_memcntl	Manages physical attachment of a file.
vnop_mkdir	Creates a directory.
vnop_mknod	Creates a file of arbitrary type.
vnop_open	Gets read and/or write access to a file.
vnop_rdwr	Reads or writes a file.
vnop_rdwr_attr	Reads or writes a file and returns attributes.
vnop_readdir	Reads directory entries in standard format.
vnop_readdir_eofp	Reads directories and returns end of file indication.
vnop_readlink	Reads the contents of a symbolic link.
vnop_rele	Releases a reference to a virtual node (v-node).
vnop_remove	Unlinks a file or directory.
vnop_rename	Renames a file or directory.
vnop_revoke	Revokes access to an object.
vnop_rmdir	Removes a directory.
vnop_seek	Moves the current offset in a file.
vnop_select	Polls a v-node for pending I/O.
vnop_setacl	Sets information about access control for a file.
vnop_setattr	Sets attributes of a file.
vnop_setxacl	Sets information about access control for a file. Provides an advanced interface compared to **vnop_setacl**.
vnop_strategy	Reads or writes blocks of a file.
vnop_symlink	Creates a symbolic link.
vnop_unmap	Destroys a file or memory association.

Related information:

Virtual File System Overview

Virtual File System Kernel Extensions Overview

vfs_aclxcntl Entry Point

Purpose

Implements access-control-specific control operations for a file system.

Syntax

```
int vfs_aclxcntl (vfsp, vp, cmd, uiop, argsize, crp)

struct vfs     *vfsp;
struct vnode   *vp;
int               cmd;
struct uio     *uiop;
size_t         *argsize;
struct ucred   *crp;
```

Description

The **vfs_aclxcntl** entry point is invoked to perform various ACL-specific control operations on the underlying physical file system. If a file system is implemented to support this interface, it needs to adhere to the various commands and arguments defined for the interface. A file system implementation can define *cmd* parameter values and corresponding control functions that are specific to the file system. The *cmd* parameter for these functions has values defined globally for all the physical file systems. These control operations can be issued with the ACL library interfaces.

Parameters

Item **Description**

vfsp Points to the file system for which the control operation is to be issued.

vp Points to the virtual node pointer to the file path of the file system for which the control operation is being requested.

cmd Specifies which control operation to perform. Has one of the following values:

ACLCNTL_GETACLXTYPES
Returns the various ACL types supported for the file system instance. This area is of the following structure type:

```
typedef struct   _acl_types_list_t  {
        uint32_t   num_entries;  // in the buffer to follow
        uint32_t   pad;          // reserved space
        acl_type_t entries[MAX_ACL_TYPES];
 // Array of ACL types
} acl_types_list_t ;
```

If the buffer space is not enough to accommodate ACL types supported by the physical file system, **errno** is set to **ENOSPC** and the necessary size of the buffer is returned in *argsize*.

ACLCNTL_GETACLXTYPEINFO
Returns the characteristics information related to an ACL type for the file system instance. This area is of the following structure type:

```
typedef struct      _acl_type_info_t  {
        acl_type_t    acl_type;
// ACL type for which info is needed
        uint8_t    acl_type_info;
// Start of ACL characteristics data
} _acl_type_info_t ;
```

acl_type_info is the start byte of the ACL-related characteristics information. ACL characteristics information depends on the ACL type. ACL characteristics for NFS4 ACL type have the following structure:

```
typedef struct      _nfs4_acl_type_info_t  {
        uint32_t  version;
// Version of this structure
        uint32_t  acl_suport;
// Support of Access control entry types.
} nfs4_acl_type_info_t ;
```

If the buffer space is not enough to accommodate the ACL types supported by the physical file system, **errno** is set to **ENOSPC** and the necessary size of the buffer is returned in *argsize*.

uiop Identifies data specific to the control operation. If the *cmd* parameter has a value of **ACLCNTL_GETACLXTYPES**, *uiop* points to a buffer area where the file system stores the supported ACL types. If the *cmd* parameter has a value of **ACLCNTL_GETACLXTYPEINFO**, *uiop* points to a buffer area where the file system stores the ACL characteristics information.

argsize Identifies the length of the data specified by the *arg* parameter. This buffer is used to return the necessary buffer size, in case the buffer size provided by the user is not enough.

crp Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Execution Environment

The **vfs_aclxcntl** entry point can be called from the process environment only.

Return Values

Upon successful completion, the **vfs_aclxcntl** entry point returns 0. Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Item	Description
EACCES	The *cmd* parameter requires a privilege that the current process does not have.
EINVAL	Indicates that the *cmd* parameter is not a supported control, or the *arg* parameter is not a valid argument for the command.
ENOSPC	The input buffer was not sufficient for storing the requested information.

Related information:

Virtual File System Overview

Virtual File System Kernel Extensions Overview

Logical File System Overview

vfs_cntl Entry Point

Purpose

Implements control operations for a file system.

Syntax

```
int vfs_cntl (vfsp, cmd, arg, argsize, crp)
struct vfs * vfsp;
int  cmd;
caddr_t  arg;
unsigned long  argsize;
struct ucred * crp;
```

Parameters

Item	Description
vfsp	Points to the file system for which the control operation is to be issued.
cmd	Specifies which control operation to perform.
arg	Identifies data specific to the control operation.
argsize	Identifies the length of the data specified by the *arg* parameter.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_cntl** entry point is invoked by the logical file system to request various control operations on the underlying file system. A file system implementation can define file system-specific *cmd* parameter values and corresponding control functions. The *cmd* parameter for these functions should have a minimum value of 32768. These control operations can be issued with the **fscntl** subroutine.

Note: The only system-supported control operation is **FS_EXTENDFS**. This operation increases the file system size and accepts an *arg* parameter that specifies the new size. The **FS_EXTENDFS** operation ignores the *argsize* parameter.

Execution Environment

The **vfs_cntl** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Non-zero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. Typical values include:

Item	Description
EINVAL	Indicates that the *cmd* parameter is not a supported control, or the *arg* parameter is not a valid argument for the command.
EACCES	Indicates that the *cmd* parameter requires a privilege that the current process does not have.

Related information:

fscntl subroutine

Virtual File System Overview

Virtual File System Kernel Extensions Overview

vfs_hold or vfs_unhold Kernel Service

Purpose

Holds or releases a **vfs** structure.

Syntax

```
#include <sys/vfs.h>

void vfs_hold(vfsp)
struct vfs *vfsp;

int vfs_unhold(vfsp)
struct vfs *vfsp;
```

Parameter

Item	Description
vfsp	Points to a **vfs** structure.

Description

The **vfs_hold** kernel service holds a **vfs** structure and the **vfs_unhold** kernel service releases it. These routines manage a use count for a virtual file system (VFS). A use count greater than 1 prevents the virtual file system from being unmounted.

Execution Environment

These kernel services can be called from the process environment only.

Return Values

The **vfs_hold** kernel service has no return value.

The **vfs_unhold** kernel service returns the original value of the hold count.

vfs_init Entry Point

Purpose

Initializes a virtual file system.

Syntax

```
int vfs_init ( gfsp)
struct gfs *gfsp;
```

Parameter

Item	Description
gfsp	Points to a file system's attribute structure.

Description

The **vfs_init** entry point is invoked to initialize a file system. It is called when a file system implementation is loaded to perform file system-specific initialization.

The **vfs_init** entry point is not called through the virtual file system switch. Instead, it is called indirectly by the **gfsadd** kernel service when the **vfs_init** entry point address is stored in the **gfs** structure passed to the **gfsadd** kernel service as a parameter. (The **vfs_init** address is placed in the `gfs_init` field of the **gfs** structure.) The **gfs** structure is defined in the **/usr/include/sys/gfs.h** file.

Note: The return value for the **vfs_init** entry point is passed back as the return value from the **gfsadd** kernel service.

Execution Environment

The **vfs_init** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related reference:

"gfsadd Kernel Service" on page 192

Related information:

Virtual File System Overview

Virtual File System Kernel Extensions Overview

vfs_mount Entry Point

Purpose

Mounts a virtual file system.

Syntax

```
int vfs_mount ( vfsp)
struct vfs *vfsp;
struct ucred * crp;
```

Parameter

Item	Description
vfsp	Points to the newly created **vfs** structure.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_mount** entry point is called by the logical file system to mount a new file system. This entry point is called after the **vfs** structure is allocated and initialized. Before this structure is passed to the **vfs_mount** entry point, the logical file system:

- Guarantees the syntax of the **vmount** or **mount** subroutines.
- Allocates the **vfs** structure.
- Resolves the stub to a virtual node (v-node). This is the `vfs_mntdover` field in the **vfs** structure.
- Initializes the following virtual file system fields:

Field	Description
`vfs_flags`	Initialized depending on the type of mount. This field takes the following values: **VFS_MOUNTOK** The user has write permission in the stub's parent directory and is the owner of the stub. **VFS_SUSER** The user has root user authority. **VFS_NOSUID** Execution of **setuid** and **setgid** programs from this mount are not allowed. **VFS_NODEV** Opens of devices from this mount are not allowed.
`vfs_type`	Initialized to the `/` (root) file system type when the **mount** subroutine is used. If the **vmount** subroutine is used, the **vfs_type** field is set to the *type* parameter supplied by the user. The logical file system verifies the existence of the *type* parameter.
`vfs_ops`	Initialized according to the `vfs_type` field.
`vfs_mntdover`	Identifies the v-node that refers to the stub path argument. This argument is supplied by the **mount** or **vmount** subroutine.
`vfs_date`	Holds the time stamp. The time stamp specifies the time to initialize the virtual file system.
`vfs_number`	Indicates the unique number sequence representing this virtual file system.
`vfs_mdata`	Initialized with the **vmount** structure supplied by the user. The virtual file system data is detailed in the **/usr/include/sys/vmount.h** file. All arguments indicated by this field are copied to kernel space.

Execution Environment

The **vfs_mount** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

mount subroutine

Virtual File System Overview

Logical File System Overview

vfs_root Entry Point

Purpose

Returns the root v-node of a virtual file system (VFS).

Syntax

```
int vfs_root ( vfsp,  vpp,  crp)
struct vfs *vfsp;
struct vnode **vpp;
struct ucred *crp;
```

Parameters

Item	Description
vfsp	Points to the **vfs** structure.
vpp	Points to the place to return the v-node pointer.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_root** entry point is invoked by the logical file system to get a pointer to the root v-node of the file system. When successful, the *vpp* parameter points to the root virtual node (v-node) and the v-node hold count is incremented.

Execution Environment

The **vfs_root** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

Virtual File System Overview

Understanding Data Structures and Header Files for Virtual File Systems

Logical File System Overview

vfs_search Kernel Service

Purpose

Searches the vfs list.

Syntax

```
int vfs_search ( vfs_srchfcn,  srchargs)
(int (*vfs_srchfcn)(struct vfs *, caddr_t);
caddr_t srchargs;
```

Parameters

Item	Description
vfs_srchfcn	Points to a search function. The search function is identified by the *vfs_srchfcn* parameter. This function is used to examine or modify an entry in the vfs list. The search function is called once for each currently active VFS. If the search function returns a value of 0, iteration through the vfs list continues to the next entry. If the return value is nonzero, **vfs_search** kernel service returns to its caller, passing back the return value from the search function. When the system invokes this function, the system passes it a pointer to a virtual file system (VFS) and the *srchargs* parameter.
srchargs	Points to data to be used by the search function. This pointer is not used by the **vfs_search** kernel service but is passed to the search function.

Description

The **vfs_search** kernel service searches the vfs list. This kernel service allows a process outside the file system to search the vfs list. The **vfs_search** kernel service locks out all activity in the vfs list during a search. Then, the kernel service iterates through the vfs list and calls the search function on each entry.

The search function must not request locks that could result in deadlock. In particular, any attempt to do lock operations on the vfs list or on other VFS structures could produce deadlock.

The performance of the **vfs_search** kernel service may not be acceptable for functions requiring quick response. Iterating through the vfs list and making an indirect function call for each structure is inherently slow.

Execution Environment

The **vfs_search** kernel service can be called from the process environment only.

Return Values

This kernel service returns the value returned by the last call to the search function.

Related Information

vfs_statfs Entry Point

Purpose

Returns virtual file system statistics.

Syntax

```
int vfs_stafs ( vfsp,  stafsp,  crp)
struct vfs *vfsp;
struct statfs *stafsp;
struct ucred *crp;
```

Parameters

Item	Description
vfsp	Points to the **vfs** structure being queried. This structure is defined in the **/usr/include/sys/vfs.h** file.
stafsp	Points to a **statfs** structure. This structure is defined in the **/usr/include/sys/statfs.h** file.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_stafs** entry point is called by the logical file system to obtain file system characteristics. Upon return, the **vfs_statfs** entry point has filled in the following fields of the **statfs** structure:

Field	Description
`f_blocks`	Specifies the number of blocks.
`f_files`	Specifies the total number of file system objects.
`f_bsize`	Specifies the file system block size.
`f_bfree`	Specifies the number of free blocks.
`f_ffree`	Specifies the number of free file system objects.
`f_fname`	Specifies a 32-byte string indicating the file system name.
`f_fpack`	Specifies a 32-byte string indicating a pack ID.
`f_name_max`	Specifies the maximum length of an object name.

Fields for which a **vfs** structure has no values are set to 0.

Execution Environment

The **vfs_statfs** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

statfs subroutine

Virtual File System Overview

Virtual File System Kernel Extensions Overview

vfs_sync Entry Point

Purpose

Requests that file system changes be written to permanent storage.

Syntax

```
int vfs_sync (* gfsp)
struct gfs *gfsp;
```

Parameter

Item	Description
gfsp	Points to a **gfs** structure. The **gfs** structure describes the file system type. This structure is defined in the **/usr/include/sys/gfs.h** file.

Description

The **vfs_sync** entry point is used by the logical file system to force all data associated with a particular virtual file system type to be written to its storage. This entry point is used to establish a known consistent state of the data.

Note: The **vfs_sync** entry point is called once per file system type rather than once per virtual file system.

Execution Environment

The **vfs_sync** entry point can be called from the process environment only.

Return Values

The **vfs_sync** entry point is advisory. It has no return values.

Related information:

sync subroutine

Virtual File System Overview

Virtual File System Kernel Extensions Overview

Logical File System Overview

vfs_umount Entry Point

Purpose

Unmounts a virtual file system.

Syntax

```
int vfs_umount ( vfsp,  crp)
struct vfs *vfsp;
struct ucred *crp;
```

Parameters

Item	Description
vfsp	Points to the **vfs** structure being unmounted. This structure is defined in the **/usr/include/sys/vfs.h** file.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_umount** entry point is called to unmount a virtual file system. The logical file system performs services independent of the virtual file system that initiate the unmounting. The logical file system services:

- Guarantee the syntax of the **uvmount** subroutine.
- Perform permission checks:
  - If the *vfsp* parameter refers to a device mount, then the user must have root user authority to perform the operation.

  - If the *vfsp* parameter does not refer to a device mount, then the user must have root user authority or write permission in the parent directory of the mounted-over virtual node (v-node), as well as write permission to the file represented by the mounted-over v-node.
- Ensure that the virtual file system being unmounted contains no mount points for other virtual file systems.
- Ensure that the root v-node is not in use except for the mount. The root v-node is also referred to as the mounted v-node.
- Clear the `v_mvfsp` field in the stub v-node. This prevents lookup operations already in progress from traversing the soon-to-be unmounted mount point.

The logical file system assumes that, if necessary, successful **vfs_umount** entry point calls free the root v-node. An error return from the **vfs_umount** entry point causes the mount point to be re-established. A 0 (zero) returned from the **vfs_umount** entry point indicates the routine was successful and that the **vfs** structure was released.

Execution Environment

The **vfs_umount** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

umount subroutine

vmount subroutine

Understanding Data Structures and Header Files for Virtual File Systems

vfs_vget Entry Point

Purpose

Converts a file identifier into a virtual node (v-node).

Syntax

```
int vfs_vget ( vfsp,  vpp,  fidp,  crp)
struct vfs *vfsp;
struct vnode **vpp;
struct fileid *fidp;
struct ucred *crp;
```

Parameters

Item	Description
vfsp	Points to the virtual file system that is to contain the v-node. Any returned v-node should belong to this virtual file system.
vpp	Points to the place to return the v-node pointer. This is set to point to the new v-node. The fields in this v-node should be set as follows: **v_vntype** The type of v-node dependent on private data. **v_count** Set to at least 1 (one). **v_pdata** If a new file, set to the private data for this file system.
fidp	Points to a file identifier. This is a file system-specific file identifier that must conform to the **fileid** structure. **Note:** If the *fidp* parameter is invalid, the *vpp* parameter should be set to a null value by the **vfs_vget** entry point.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vfs_vget** entry point is called to convert a file identifier into a v-node. This entry point uses information in the *vfsp* and *fidp* parameters to create a v-node or attach to an existing v-node. This v-node represents, logically, the same file system object as the file identified by the *fidp* parameter.

If the v-node already exists, successful operation of this entry point increments the v-node use count and returns a pointer to the v-node. If the v-node does not exist, the **vfs_vget** entry point creates it using the **vn_get** kernel service and returns a pointer to the new v-node.

Execution Environment

The **vfs_vget** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. A typical value includes:

Item	Description
EINVAL	Indicates that the remote virtual file system specified by the *vfsp* parameter does not support chained mounts.

Related reference:

"vn_get Kernel Service" on page 579

Related information:

access subroutine

Virtual File System Overview

vnop_access Entry Point

Purpose

Requests validation of user access to a virtual node (v-node).

Syntax

```
int vnop_access ( vp,  mode,  who,  crp)
struct vnode *vp;
```

```
int mode;
int who;
struct ucred *crp;
```

Parameters

Item	Description
vp	Points to the v-node.
mode	Identifies the access mode.
who	Specifies the IDs for which to check access. This parameter should be one of the following values, which are defined in the **/usr/include/sys/access.h** file: **ACC_SELF** Determines if access is permitted for the current process. The effective user and group IDs and the supplementary group ID of the current process are used for the calculation. **ACC_ANY** Determines if the specified access is permitted for any user, including the object owner. The *mode* parameter must contain only one of the valid modes. **ACC_OTHERS** Determines if the specified access is permitted for any user, excluding the owner. The *mode* parameter must contain only one of the valid modes. **ACC_ALL** Determines if the specified access is permitted for all users. (This is a useful check to make when files are to be written blindly across networks.) The *mode* parameter must contain only one of the valid modes.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_access** entry point is used by the logical volume file system to validate access to a v-node. This entry point is used to implement the **access** subroutine. The v-node is held for the duration of the **vnop_access** entry point. The v-node count is unchanged by this entry point.

In addition, the **vnop_access** entry point is used for permissions checks from within the file system implementation. The valid types of access are listed in the **/usr/include/sys/access.h** file. Current modes are read, write, execute, and existence check.

Note: The **vnop_access** entry point must ensure that write access is not requested on a read-only file system.

Execution Environment

The **vnop_access** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. A typical value includes:

Item	Description
EACCES	Indicates no access is allowed.

Related information:

access subroutine

Virtual File System Overview

Virtual File System Kernel Extensions Overview

vnop_close Entry Point

Purpose

Closes a file associated with a v-node (virtual node).

Syntax

```
int vnop_close ( vp,  flag,  vinfo,  crp)
struct vnode *vp;
int flag;
caddr_t vinfo;
struct ucred *crp;
```

Parameters

Item	Description
vp	Points to the v-node.
flag	Identifies the flag word from the file pointer.
vinfo	This parameter is not used.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_close** entry point is used by the logical file system to announce that the file associated with a given v-node is now closed. The v-node continues to remain active but will no longer receive read or write requests through the **vnop_rdwr** entry point.

A **vnop_close** entry point is called only when the use count of an associated file structure entry goes to 0 (zero).

Note: The v-node is held over the duration of the **vnop_close** entry point.

Execution Environment

The **vnop_close** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Note: The **vnop_close** entry point may fail and an error will be returned to the application. However, the v-node is considered closed.

Related reference:

"vnop_open Entry Point" on page 674

Related information:

close subroutine

Virtual File System Overview

vnop_create Entry Point

Purpose

Creates a new file.

Syntax

```
int vnop_create (dp, vpp, flag, pname, mode, vinfop, crp)
struct vnode * dp;
struct vnode ** vpp;
int  flag;
char * pname;
int  mode;
caddr_t * vinfop;
struct ucred * crp;
```

Parameters

Item	Description
dp	Points to the virtual node (v-node) of the parent directory.
vpp	Points to the place in which the pointer to a v-node for the newly created file is returned.
flag	Specifies an integer flag word. The **vnop_create** entry point uses this parameter to open the file.
pname	Points to the name of the new file.
mode	Specifies the mode for the new file.
vinfop	This parameter is unused.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_create** entry point is invoked by the logical file system to create a regular (v-node type **VREG**) file in the directory specified by the *dp* parameter. (Other v-node operations create directories and special files.) Virtual node types are defined in the **/usr/include/sys/vnode.h** file. The v-node of the parent directory is held during the processing of the **vnop_create** entry point.

To create a file, the **vnop_create** entry point does the following:

- Opens the newly created file.
- Checks that the file system associated with the directory is not read-only.

  Note: The logical file system calls the **vnop_lookup** entry point before calling the **vnop_create** entry point.

Execution Environment

The **vnop_create** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related reference:

"vnop_lookup Entry Point" on page 669

Related information:

Virtual File System Overview

Virtual File System Kernel Extensions Overview

vnop_create_attr Entry Point

Purpose

Creates a new file.

Syntax

int vnop_create_attr *(dvp, vpp, flags, name, vap, vcf, finfop, crp)* **struct vnode** **dvp*; **struct vnode** **vpp*; **int flags**; **char** **name*; **struct vattr** **vap*; **int** *vcf*; **caddr_t** *finfop*; **struct ucred** **crp*;

Parameters

Item	Description
dvp	Points to the directory vnode.
vpp	Points to the newly created vnode pointer.
flags	Specifies file creation flags.
name	Specifies the name of the file to create.
vattr	Points to the initial attributes.
vcf	Specifies create flags.
finfop	Specifies address of finfo field.
crp	Specifies user's credentials.

Description

The **vnop_create_attr** entry point is used to create a new file. This operation is similar to the vnop_create entry point except that the initial file attributes are passed in a vattr structure.

The va_mask field in the vattr structure identifies which attributes are to be applied. For example, if the AT_SIZE bit is set, then the file system should use va_size for the initial file size. For all vnop_create_attr calls, at least AT_TYPE and AT_MODE must be set.

The vcf parameter controls how the new vnode is to be activated. If vcf is set to VC_OPEN, then the new object should be opened. If vcf is VC_LOOKUP, then the new object should be created, but not opened. If vcf is VC_DEFAULT, then the new object should be created, but the vnode for the object is not activated.

File systems that do not define GFS_VERSION421 in their gfs flags do not need to supply a vnop_create_attr entry point. The logical file system will funnel all creation requests through the old vnop_create entry point.

Execution Environment

The **vnop_create_attr** entry point can be called from the process environment only.

Return Values

Item	Description
Zero	Indicates a successful operation; **vpp* contains a pointer to the new vnode.
Nonzero	Indicates that the operation failed; return values should be chosen from the **/usr/include/sys/errno.h** file.

vnop_fclear Entry Point

Purpose

Releases portions of a file.

Syntax

```
int vnop_fclear (vp, flags, offset, len, vinfo, crp)
struct vnode * vp;
int  flags;
offset_t  offset;
offset_t  len;
caddr_t  vinfo;
struct ucred * crp;
```

Parameters

Item	Description
vp	Points to the virtual node (v-node) of the file.
flags	Identifies the flags from the open file structure.
offset	Indicates where to start clearing in the file.
len	Specifies the length of the area to be cleared.
vinfo	This parameter is unused.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_fclear** entry point is called from the logical file system to clear bytes in a file, returning whole free blocks to the underlying file system. This entry point performs the clear regardless of whether the file is mapped.

Upon completion of the **vnop_fclear** entry point, the logical file system updates the file offset to reflect the number of bytes cleared. Also upon completion, if either the starting or ending offset is past the starting end of file, the file is extended.

Execution Environment

The **vnop_fclear** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

fclear subroutine

Virtual File System Overview

vnop_fid Entry Point

Purpose

Builds a file identifier for a virtual node (v-node).

Syntax

```
int vnop_fid ( vp,  fidp,  crp)
struct vnode *vp;
struct fileid *fidp;
struct ucred *crp;
```

Parameters

Item	Description
vp	Points to the v-node that requires the file identifier.
fidp	Points to where to return the file identifier.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_fid** entry point is invoked to build a file identifier for the given v-node. This file identifier must contain sufficient information to find a v-node that represents the same file when it is presented to the **vfs_get** entry point.

Execution Environment

The **vnop_fid** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

Virtual File System Overview

Virtual File System Kernel Extensions Overview

Logical File System Overview

vnop_finfo Entry Point

Purpose

Returns information about a file.

Syntax

int vnop_finfo *(vp, cmd, bufp, length, crp)* **struct vnode** **vp*; **int** *cmd*; **void** **bufp*; **int** *length*; **struct ucred** **crp*;

Parameters

Item	Description
vp	Points to the vnode to be queried.
cmd	Specifies the command parameter.
bufp	Points to the buffer for the information.
length	Specifies the length of the buffer.
crp	Specifies user's credentials.

Description

The **vnop_finfo** entry point is used to query a file system. It is used primarily to implement the **pathconf** and **fpathconf** subroutines. The **command** parameter defines what type of query is being done. The query commands and the associated data structures are defined in **<sys/finfo.h>**. If the file system does not support the particular query, it should return ENOSYS.

File systems that do not define GFS_VERSION421 in their gfs flags do not need to supply a **vnop_finfo** entry point. If the command is FI_PATHCONF, then the logical file system returns generic pathconf information. If the query is other than FI_PATHCONF, then the request fails with EINVAL.

Execution Environment

The **vnop_finfo** entry point can be called from the process environment only.

Return Values

Item	Description
Zero	Indicates a successful operation.
Nonzero	Indicates that the operation failed; return values should be chosen from the **/usr/include/sys/errno.h** file.

Related information:

pathconf, fpathconf

Virtual File System Overview

Logical File System Overview

vnop_fsync, vnop_fsync_range Entry Points

Purpose

Flushes file data from memory to disk.

Syntax

```
int vnop_fsync ( vp, flags, vinfo, crp)
struct vnode *vp;
long flags;
long vinfo;
struct ucred *crp;

int vnop_fsync_range ( vp, flags, vinfo, offset, length, crp)
struct vnode *vp;
```

```
long flags;
long vinfo;
offset_t offset;
offset_t length;
struct ucred *crp;
```

Parameters

Item	Description
vp	Points to the virtual node (v-node) of the file.
flags	Identifies flags from the open file and the flags that govern the action to be taken. It can be one of the following values: **FDATASYNC** The changed data in the range specified by the `offset` and `length` parameters is written to the storage. If the metadata of the file is changed and this changed metadata must read the data, the metadata is also written to the storage. Otherwise, the metadata is not updated. **FFILESYNC** The changed data in the range specified by the `offset` and `length` parameters is written to the storage. If any metadata is changed, all of the changed user data is written to the storage. Metadata changes and file attributes including time stamps are also written to the storage. **FNOCACHE** The changed data is written to the storage similar to the `FDATASYNC` flag value. The full pages in the range specified by the `offset` and `length` parameters are removed from the memory cache. The pages are removed from the cache even if the pages are not changed. This operation is also applicable to the files that are open only for reading.
vinfo	This parameter is currently not used.
offset	Specifies the starting offset in the file of the data to be flushed.
length	Specifies the length of the data to be flushed. If you specify the value as zero, all cached data is flushed.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_fsync** entry point is called by the logical file system to request that all modifications associated with a given v-node to be flushed out to permanent storage. This must be done synchronously so that the caller can assure that all I/O has completed successfully. The **vnop_fsync_range** entry point provides the same function but limits the data to be written to a specified range in the file.

Execution Environment

The **vnop_fsync** and **vnop_fsync_range** entry points can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

fsync subroutine

Virtual File System Kernel Extensions Overview

Logical File System Overview

vnop_ftrunc Entry Point

Purpose

Truncates a file.

Syntax

```
int vnop_ftrunc (vp, flags, length, vinfo, crp)
struct vnode * vp;
int  flags;
offset_t  length;
caddr_t  vinfo;
struct ucred * crp;
```

Parameters

Item	Description
vp	Points to the virtual node (v-node) of the file.
flags	Identifies flags from the open file structure.
length	Specifies the length to which the file should be truncated.
vinfo	This parameter is unused.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_ftrunc** entry point is invoked by the logical file system to decrease the length of a file by truncating it. This operation is unsuccessful if any process other than the caller has locked a portion of the file past the specified offset.

Execution Environment

The **vnop_ftrunc** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

ftruncate subroutine

Virtual File System Overview

Logical File System Overview

vnop_getacl Entry Point

Purpose

Retrieves the access control list (ACL) for a file.

Syntax

```
#include <sys/acl.h>
```

```
int vnop_getacl ( vp,  uiop,  crp)
struct vnode *vp;
struct uio *uiop;
struct ucred *crp;
```

Description

The **vnop_getacl** entry point is used by the logical file system to retrieve the access control list (ACL) for a file to implement the **getacl** subroutine.

Parameters

Item	Description
vp	Specifies the virtual node (v-node) of the file system object.
uiop	Specifies the **uio** structure that defines the storage for the ACL.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Execution Environment

The **vnop_getacl** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates a successful operation.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. A valid value includes:

Item	Description
ENOSPC	Indicates that the buffer size specified in the *uiop* parameter was not large enough to hold the ACL. If this is the case, the first word of the user buffer (data in the **uio** structure specified by the *uiop* parameter) is set to the appropriate size.

Related reference:

"uio Structure" on page 638

Related information:

chacl subroutine

statacl subroutine

Virtual File System Overview

vnop_getattr Entry Point

Purpose

Gets the attributes of a file.

Syntax

```
int vnop_getattr ( vp,  vap,  crp)
struct vnode *vp;
struct vattr *vap;
struct ucred *crp;
```

Parameters

Item	Description
vp	Specifies the virtual node (v-node) of the file system object.
vap	Points to a **vattr** structure.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_getattr** entry point is called by the logical file system to retrieve information about a file. The **vattr** structure indicated by the *vap* parameter contains all the relevant attributes of the file. The **vattr** structure is defined in the **/usr/include/sys/vattr.h** file. This entry point is used to implement the **stat**, **fstat**, and **lstat** subroutines.

Note: The indicated v-node is held for the duration of the **vnop_getattr** subroutine.

Execution Environment

The **vnop_getattr** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

statx subroutine

Virtual File System Overview

Virtual File System Kernel Extensions Overview

vnop_getxacl Entry Point

Purpose

Retrieves the access control list (ACL) for a file. This is an advanced version of **vnop_getacl** interface.

Syntax

```
#include <sys/acl.h>
int vnop_getxacl (vp, ctl_flags, acl_type, uiop, acl_len, mode_info, crp)

struct vnode   *vp;
uint64_t       ctl_flags;
acl_type_t     *acl_type;
struct uio     *uiop;
size_t         *acl_len;
mode_t         *mode_info;
struct ucred   *crp;
```

Description

The **vnop_getxacl** entry point retrieves the access control list (ACL) for a file system object. It is an advanced version of **vnop_getacl** interface and provides for ACL-type-based operations. Note that this interface can be used to obtain the ACL type and length information, without actually retrieving the ACL data (see the *ctl_flags* description for more details).

Parameters

Item	Description
vp	Specifies the virtual node (v-node) of the file system object.
acl_type	Points to buffer space for file systems to return the ACL type associated with the file system object. The value should normally be set to **ACL_ANY** or 0 when the call is made. Some physical file systems can solicit ACL requests for a particular ACL type. In such cases, the caller provides the ACL type requested in this buffer. **Note:** The latter issue is file system implementation specific. For example, when ACL information is requested with an input ACL type, a physical file system might return an error if the existing ACL associated with the file system object is of a different ACL type. Or, the file system might emulate an ACL of the type requested and return.
acl_len	Pointer to a *length* variable. The space pointed to is used as an input, as well as output, parameter. As input, the value will indicate the size of buffer *uiop*. When the call returns, this space holds the actual length of the ACL (true for when the call is successful or when the call fails with **errno** set to **ENOSPC**).
ctl_flags	A 64-bit bit mask that provides control over the ACL retrieval and for any future variations in the interface. The following value is defined for these flags: **GET_ACLINFO_ONLY** Gets only the ACL type and length information from the underlying file system. When this bit is set, arguments such as *mode_info* can be set to NULL. All other cases must be valid buffer pointers or else an error is returned. If this bit is not specified, all the other information about the ACL (such as ACL data and mode information) is returned.
uiop	Specifies the **uio** structure that provides space for the store of the ACL.
mode_info	This value indicates any mode word information that needs to be retrieved for the file system object as part of this ACL get operation.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Execution Environment

The **vnop_getxacl** entry point can be called from the process environment only.

Return Values

Upon successful completion, the **vnop_getxacl** entry point returns 0. Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Item	Description
ENOSPC	Indicates that the buffer size specified in the *uiop* parameter was not large enough to hold the ACL.

Note: This list of error numbers is not complete and is dependent on the particular physical file system implementation supporting the ACL.

Related reference:

"uio Structure" on page 638

Related information:

chacl subroutine

Virtual File System Overview

vnop_hold Entry Point

Purpose

Assures that a virtual node (v-node) is not destroyed.

Syntax

```
int vnop_hold ( vp)
struct vnode *vp;
```

Parameter

Item	Description
vp	Points to the v-node.

Description

The **vnop_hold** entry point increments the `v_count` field, the hold count on the v-node, and the v-node's underlying g-node (generic node). This incrementation assures that the v-node is not deallocated.

Execution Environment

The **vnop_hold** entry point can be called from the process environment only.

Return Values

The **vnop_hold** entry point cannot fail and therefore has no return values.

Related information:

Virtual File System Overview

vnop_ioctl Entry Point

Purpose

Requests I/O control operations on special files.

Syntax

```
int vnop_ioctl (vp, cmd, arg, flags, ext, crp)
struct vnode * vp;
int  cmd;
caddr_t  arg;
int  flags,  ext;
struct ucred * crp;
```

Parameters

Item	Description
vp	Points to the virtual node (v-node) on which to perform the operation.
cmd	Identifies the specific command. Common operations for the **ioctl** subroutine are defined in the **/usr/include/sys/ioctl.h** file. The file system implementation can define other **ioctl** operations.
arg	Defines a command-specific argument. This parameter can be a single word or a pointer to an argument (or result structure).
flags	Identifies flags from the open file structure.
ext	Specifies the extended parameter passed by the **ioctl** subroutine. The **ioctl** subroutine always sets the *ext* parameter to 0.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_ioctl** entry point is used by the logical file system to perform miscellaneous operations on special files. If the file system supports special files, the information is passed down to the **ddioctl** entry point of the device driver associated with the given v-node.

Execution Environment

The **vnop_ioctl** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. A valid value includes:

Item	Description
EINVAL	Indicates the file system does not support the entry point.

Related information:

ioctl subroutine

Logical File System Overview

vnop_link Entry Point

Purpose

Requests a hard link to a file.

Syntax

```
int vnop_link ( vp, dp, name, crp)
struct vnode *vp;
struct vnode *dp;
caddr_t *name;
struct ucred *crp;
```

Parameters

Item	Description
vp	Points to the virtual node (v-node) to link to. This v-node is held for the duration of the linking process.
dp	Points to the v-node for the directory in which the link is created. This v-node is held for the duration of the linking process.
name	Identifies the new name of the entry.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_link** entry point is invoked to create a new hard link to an existing file as part of the link subroutine. The logical file system ensures that the *dp* and *vp* parameters reside in the same virtual file system, which is not read-only.

Execution Environment

The **vnop_link** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

Virtual File System Overview

Virtual File System Kernel Extensions Overview

Logical File System Overview

vnop_lockctl Entry Point

Purpose

Sets, checks, and queries record locks.

Syntax

```
int vnop_lockctl (vp, offset, lckdat, cmd, retry_fn, retry_id, crp)
struct vnode * vp;
offset_t  offset;
struct eflock * lckdat;
int  cmd;
int (* retry_fn)();
caddr_t  retry_id;
struct ucred * crp;
```

Parameters

Item	Description
vp	Points to the file's virtual node (v-node).
offset	Indicates the file offset from the open file structure. This parameter is used to establish where the lock region begins.
lckdat	Points to the **elock** structure. This structure describes the lock operation to perform.
cmd	Identifies the type of lock operation the **vnop_lockctl** entry point is to perform. It is a bit mask that takes the following lock-control values: **SETFLCK** If set, performs a lock set or clear. If clear, returns the lock information. The `l_type` field in the **eflock** structure indicates whether a lock is set or cleared. **SLPFLCK** If the lock is unavailable immediately, wait for it. This is only valid when the **SETFLCK** flag is set.
retry_fn	Points to a subroutine that is called when a lock is retried. This subroutine is not used if the lock is granted immediately. **Note:** If the *retry_fn* parameter is not a null value, the **vnop_lockctl** entry point will not sleep, regardless of the **SLPFLCK** flag.
retry_id	Points to the location where a value can be stored. This value can be used to correlate a retry operation with a specific lock or set of locks. The retry value is only used in conjunction with the *retry_fn* parameter. **Note:** This value is an opaque value and should not be used by the caller for any purpose other than a lock correlation. (This value should not be used as a pointer.)
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_lockctl** entry point is used to request record locking. This entry point uses the information in the **eflock** structure to implement record locking.

If a requested lock is blocked by an existing lock, the **vnop_lockctl** entry point should establish a sleeping lock with the retry subroutine address (specified by the *retry_fn* parameter) stored in the entry point. The **vnop_lockctl** entry point then returns a correlating ID value to the caller (in the *retry_id* parameter), along with an exit value of **EAGAIN**. When the sleeping lock is later awakened, the retry subroutine is called with the *retry_id* parameter as its argument.

eflock Structure

The **eflock** structure is defined in the **/usr/include/sys/flock.h** file and includes the following fields:

Field	Description
l_type	Specifies type of lock. This field takes the following values: **F_RDLCK** Indicates read lock. **F_WRLCK** Indicates write lock. **F_UNLCK** Indicates unlock this record. A value of **F_UNLCK** starting at 0 until 0 for a length of 0 means unlock all locks on this file. Unlocking is done automatically when a file is closed.
l_whence	Specifies location that the l_start field offsets.
l_start	Specifies offset from the l_whence field.
l_len	Specifies length of record. If this field is 0, the remainder of the file is specified.
l_vfs	Specifies virtual file system that contains the file.
l_sysid	Specifies value that uniquely identifies the host for a given virtual file system. This field must be filled in before the call to the **vnop_lockctl** entry point.
l_pid	Specifies process ID (PID) of the lock owner. This field must be filled in before the call to the **vnop_lockctl** entry point.

Execution Environment

The **vnop_lockctl** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. Valid values include:

Item	Description
EAGAIN	Indicates a blocking lock exists and the caller did not use the **SLPFLCK** flag to request that the operation sleep.
ERRNO	Returns an error number from the **/usr/include/sys/errno.h** file on failure.

Related information:

Virtual File System Overview

Logical File System Overview

vnop_lookup Entry Point

Purpose

Returns a v-node for a given name in a directory.

Syntax

```
int vnop_lookup (dvp, vpp, name, vattrp , crp)
struct vnode * dvp;
struct vnode ** vpp;
char * name;
struct vattr * vattrp;
struct ucred * crp;
```

Parameters

Item	Description
dvp	Points to the virtual node (v-node) of the directory to be searched. The logical file system verifies that this v-node is of a VDIR type.
name	Points to a null-terminated character string containing the file name to look up.
vattrp	Points to a **vattr** structure. If this pointer is NULL, no action is required of the file system implementation. If it is not NULL, the attributes of the file specified by the *name* parameter are returned at the address passed in the *vattrp* parameter.
vpp	Points to the place to which to return the v-node pointer, if the pointer is found. Otherwise, a null character should be placed in this memory location.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_lookup** entry point is invoked by the logical file system to find a v-node. It is used by the kernel to convert application-given path names to the v-nodes that represent them.

The use count in the v-node specified by the *dvp* parameter is incremented for this operation, and it is not decremented by the file system implementation.

If the name is found, a pointer to the desired v-node is placed in the memory location specified by the *vpp* parameter, and the v-node hold count is incremented. (In this case, this entry point returns 0.) If the file name is not found, a null character is placed in the *vpp* parameter, and the function returns a **ENOENT** value. Errors are reported with a return code from the **/usr/include/sys/errno.h** file. Possible errors are usually specific to the particular virtual file system involved.

Execution Environment

The **vnop_lookup** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

Virtual File System Overview

Virtual File System Kernel Extensions Overview

Logical File System Overview

vnop_map Entry Point

Purpose

Validates file mapping requests.

Syntax

```
int vnop_map (vp, addr, length, offset, flags, crp)
struct vnode * vp;
caddr_t  addr;
uint  length;
uint  offset;
uint  flags;
struct ucred * crp;
```

Parameters

Note: The *addr*, *offset*, and *length* parameters are unused in the current implementation. The file system is expected to store the segment ID with the file in the `gn_seg` field of the g-node for the file.

Item	Description
vp	Points to the virtual node (v-node) of the file.
addr	Identifies the location within the process address space where the mapping is to begin.
length	Specifies the maximum size to be mapped.
offset	Specifies the location within the file where the mapping is to begin.
flags	Identifies what type of mapping to perform. This value is composed of bit values defined in the **/usr/include/sys/shm.h** file. The following values are of particular interest to file system implementations: **SHM_RDONLY** The virtual memory object is read-only. **SHM_COPY** The virtual memory object is copy-on-write. If this value is set, updates to the segment are deferred until an **fsync** operation is performed on the file. If the file is closed without an **fsync** operation, the modifications are discarded. The application that called the **vnop_map** entry point is also responsible for calling the **vnop_fsync** entry point. **Note:** Mapped segments do not reflect modifications made to a copy-on-write segment.
crp	Points to the **cred** structure. This structure contains data that applications can use to validate access permission.

Description

The **vnop_map** entry point is called by the logical file system to validate mapping requests resulting from the **mmap** or **shmat** subroutines. The logical file system creates the virtual memory object (if it does not already exist) and increments the object's use count.

Execution Environment

The **vnop_map** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related reference:

"vnop_fsync, vnop_fsync_range Entry Points" on page 659

Related information:

shmat subroutine

Virtual File System Kernel Extensions Overview

vnop_map_lloff Entry Point

Purpose

Announces intention to map a file.

Syntax

int vnop_map_lloff *(vp, addr, length, offset, mflags, fflags, crp)* **struct vnode** **vp*; **caddr_t** *addr*; **offset_t** *length*; **offset_t** *offset*; **int** *mflags*; **int** *fflags*; **struct ucred** **crp*;

Parameters

Item	Description
vp	Points to the vnode to be queried.
addr	Unused.
length	Specifies the length of the mapping request.
offset	Specifies the starting offset for the map request.
mflags	Specifies the mapping flags.
fflags	Specifies the file flags.
crp	Specifies user's credentials.

Description

The **vnop_map_lloff** entry point is used to tell the file system that the file is going to be accessed by memory mapped loads and stores. The file system should fail the request if it does not support memory mapping. This interface allows applications to specify starting offsets that are larger than 2 gigabytes.

File systems that do not define GFS_VERSION421 in their gfs flags do not need to supply a **vnop_map_lloff** entry point.

Execution Environment

The **vnop_map_lloff** entry point can be called from the process environment only.

Return Values

Item	Description
Zero	Indicates a successful operation.
Nonzero	Indicates that the operation failed; return values should be chosen from the **/usr/include/sys/errno.h** file.

Related information:

shmat subroutine

mmap subroutine

Virtual File System Kernel Extensions Overview

vnop_memcntl Entry Point

Purpose

Changes or queries the physical attachment of a file.

Syntax

```
#include <sys/vnode.h>
#include <sys/fcntl.h>

int vnop_memcntl (vnode, cmd, arg, crp)
struct gnode * vnode;
int cmd;
void * arg;
struct ucred * crp;
```

Parameters

Item	Description
vnode	Points to the virtual node of the file
cmd	Specifies the operation to be performed. The *cmd* parameter can be one of the following values: • **F_ATTACH** • **F_DETACH** • **F_ATTINFO**
arg	Points to a structure containing the attach_desc_t, detach_desc_t or attinfo_desc_t information according to the specified *cmd* parameter. **F_ATTACH** attach_desc_t **F_DETACH** detach_desc_t **F_ATTINFO** attinfo_desc_t
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_memcntl** entry point requests memory attachment operations as specified by the *cmd* parameter. The *cmd* parameter determines the *arg* structure.

Execution Environment

The **vnop_memcntl** entry point can be called from the process environment only.

Return Values

Item	Description
0	Success.
non-zero	Failure.

Related information:

Workload management

vnop_mkdir Entry Point

Purpose

Creates a directory.

Syntax

```
int vnop_mkdir ( dp,  name,  mode,  crp)
struct vnode *dp;
caddr_t name;
int mode;
struct ucred *crp;
```

Parameters

Item	Description
dp	Points to the virtual node (v-node) of the parent directory of a new directory. This v-node is held for the duration of the entry point.
name	Specifies the name of a new directory.
mode	Specifies the permission modes of a new directory.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_mkdir** entry point is invoked by the logical file system as the result of the **mkdir** subroutine. The **vnop_mkdir** entry point is expected to create the named directory in the parent directory associated with the *dp* parameter. The logical file system ensures that the *dp* parameter does not reside on a read-only file system.

Execution Environment

The **vnop_mkdir** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

mkdir subroutine

Virtual File System Overview

Logical File System Overview

vnop_mknod Entry Point

Purpose

Creates a special file.

Syntax

```
int vnop_mknod (dvp, name, mode, dev, crp)
struct vnode * dvp;
caddr_t * name;
int  mode;
dev_t  dev;
struct ucred * crp;
```

Parameters

Item	Description
dvp	Points to the virtual node (v-node) for the directory to contain the new file. This v-node is held for the duration of the **vnop_mknod** entry point.
name	Specifies the name of a new file.
mode	Identifies the integer mode that indicates the type of file and its permissions.
dev	Identifies an integer device number.
crp	Points to the **cred** structure. This structure contains data that applications can use to validate access permission.

Description

The **vnop_mknod** entry point is invoked by the logical file system as the result of a **mknod** subroutine. The underlying file system is expected to create a new file in the given directory. The file type bits of the *mode* parameter indicate the type of file (regular, character special, or block special) to be created. If a special file is to be created, the *dev* parameter indicates the device number of the new special file.

The logical file system verifies that the *dvp* parameter does not reside in a read-only file system.

Execution Environment

The **vnop_mknod** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

mknod subroutine

Virtual File System Overview

Logical File System Overview

vnop_open Entry Point

Purpose

Requests that a file be opened for reading or writing.

Syntax

```
int vnop_open (vp, flag, ext, vinfop, crp)
struct vnode * vp;
int  flag;
caddr_t  ext;
caddr_t  vinfop;
struct ucred * crp;
```

Parameters

Item	Description
vp	Points to the virtual node (v-node) associated with the desired file. The v-node is held for the duration of the open process.
flag	Specifies the type of access. Access modes are defined in the **/usr/include/sys/fcntl.h** file. **Note:** The **vnop_open** entry point does not use the FCREAT mode.
ext	Points to external data. This parameter is used if the subroutine is opening a device.
vinfop	This parameter is not currently used.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_open** entry point is called to initiate a process access to a v-node and its underlying file system object. The operation of the **vnop_open** entry point varies between virtual file system (VFS) implementations. A successful **vnop_open** entry point must leave a v-node count of at least 1.

The logical file system ensures that the process is not requesting write access (with the FWRITE or FTRUNC mode) to a read-only file system.

Execution Environment

The **vnop_open** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related reference:

“vnop_close Entry Point” on page 654

Related information:

open subroutine

Virtual File System Overview

vnop_rdwr, vnop_rdwr_attr Entry Points

Purpose

Requests file I/O.

Syntax

```
int vnop_rdwr (vp, op, flags, uiop, ext, vinfo, vattrp, crp)
struct vnode * vp;
```

```
enum uio_rw op;
int flags;
struct uio * uiop;
int ext;
caddr_t vinfo;
struct vattr * vattrp;
struct ucred * crp;

int vnop_rdwr_attr (vp, op, flags, uiop, ext, vinfo, vpre, vpost, crp)
struct vnode * vp;
enum uio_rw op;
long flags;
struct uio * uiop;
ext_t ext;
caddr_t vinfo;
struct vattr * vpre;
struct vattr * vpost;
struct ucred * crp;
```

Parameters

Item	Description
vp	Points to the virtual node (v-node) of the file.
op	Specifies a number that indicates a read or write operation. This parameter has a value of either **UIO_READ** or **UIO_WRITE**. These values are found in the **/usr/include/sys/uio.h** file.
flags	Identifies flags from the open file structure.
uiop	Points to a **uio** structure. This structure describes the count, data buffer, and other I/O information.
ext	Provides an extension for special purposes. Its use and meaning are specific to virtual file systems, and it is usually ignored except for devices.
vinfo	This parameter is currently not used.
vattrp	Points to a **vattr** structure. If this pointer is NULL, no action is required of the file system implementation. If it is not NULL, the attributes of the file specified by the *vp* parameter are returned at the address passed in the *vattrp* parameter.
vpre	Points to an attributes structure for pre-operation attributes.
vpost	Points to an attributes structure for post-operation attributes.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_rdwr** entry point is used to request that data to be read or written from an object represented by a v-node. The **vnop_rdwr** entry point does the indicated data transfer and sets the number of bytes *not* transferred in the `uio_resid` field. This field is 0 (zero) on successful completion.

The **vnop_rdwr_attr** kernel service performs the same function as the **vnop_rdwr** kernel service but also allows the caller to retrieve attributes of the object either before the I/O, after or both.

Execution Environment

The **vnop_rdwr** and **vnop_rdwr_attr** entry points can be called from the process environment only.

Return Values

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. The **vnop_rdwr** entry point returns an error code if an operation did not transfer all the data requested. The only exception is if an end of file is reached on a read request. In this case, the operation still returns 0.

Related reference:

"vnop_create Entry Point" on page 655

"vnop_open Entry Point" on page 674

Related information:

Virtual File System Kernel Extensions Overview

vnop_readdir Entry Point

Purpose

Reads directory entries in standard format.

Syntax

```
int vnop_readdir ( vp,  uiop,  crp)
struct vnode *vp;
struct uio *uiop;
struct ucred *crp;
```

Parameters

Item	Description
vp	Points to the virtual node (v-node) of the directory.
uiop	Points to the **uio** structure that describes the data area into which to put the block of **dirent** structures. The starting directory offset is found in the `uiop->uio_offset` field and the size of the buffer area is found in the `uiop->uio_resid` field.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_readdir** entry point is used to access directory entries in a standard way. These directories should be returned as an array of **dirent** structures. The **/usr/include/sys/dir.h** file contains the definition of a **dirent** structure.

The **vnop_readdir** entry point does the following:

- Copies a block of directory entries into the buffer specified by the *uiop* parameter.
- Sets the `uiop->uio_resid` field to indicate the number of bytes read.

The End-of-file character should be indicated by not reading any bytes (not by a partial read). This provides directories with the ability to have some hidden information in each block.

The virtual file system-specific implementation is also responsible for setting the `uio_offset` field to the offset of the next whole block to be read.

Notes:

- If the call is meant for a JFS2 filesystem, extra processing is needed to avoid duplicate entries being returned in the user data area. The caller can check the VFS type from the directory `vnode`.
- The caller must allocate a two-element array of type `struct` **iovec** to pass with the **uio** structure. The first element is initialized to point to the user data area to receive the **dirent** structures. If the file pointer of the directory has a non-NULL `f_vinfo` field, the second `iovec` element is initialized to point to the `f_vinfo` field and the length is set to 0; the number of elements in the **uio** structure is set to 2. If the `f_vinfo` field is NULL, then the number of elements in the **uio** structure is set to 1 and the second `iovec` element remains uninitialized.
- If the caller does not have access to the directory file pointer, a **dirent** structure can be allocated in place of the `f_vinfo` field. The caller must not change this allocated structure between calls to the **vnop_readdir** entry point.

Execution Environment

The **vnop_readdir** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

readdir subroutine

Virtual File System Overview

Logical File System Overview

vnop_readdir_eofp Entry Point

Purpose

Returns directory entries.

Syntax

int vnop_readdirr_eofp *(vp, uiop, eofp, crp)* **struct vnode** **vp*; **struct uio** **uiop*; **int** **eofp*; **struct ucred** **crp*;

Parameters

Item	Description
vp	Points to the directory vnode to be processed.
uiop	Points to the uiop structure describing the user's buffer.
eofp	Points to a word that places the eop structure.
crp	Specifies user's credentials.

Description

The **vnop_readdir_eofp** entry point is used to read directory entries. It is similar to **vnop_readdir** except that it takes the additional parameter, *eofp*. The location pointed to by the *eofp* parameter should be set to 1 if the readdir request reached the end of the directory. Otherwise, it should be set to 0.

File systems that do not define GFS_VERSION421 in their gfs flags do not need to supply a **vnop_readdir_eofp** entry point.

Note: If the call is meant for a JFS2 file system, extra processing is needed to avoid duplicate entries being returned in the user data area, similar to the vnop_readdir entry point.

Execution Environment

The **vnop_readdir_eofp** entry point can be called from the process environment only.

Return Values

Item	Description
Zero	Indicates a successful operation.
Nonzero	Indicates that the operation failed; return values should be chosen from the **/usr/include/sys/errno.h** file.

Related reference:

"vnop_readdir Entry Point" on page 677

Related information:

readdir subroutine

Virtual File System Overview

Logical File System Overview

vnop_readlink Entry Point

Purpose

Reads the contents of a symbolic link.

Syntax

```
int vnop_readlink ( vp,  uio,  crp)
struct vnode *vp;
struct uio *uio;
struct ucred *crp;
```

Parameters

Item	Description
vp	Points to a virtual node (v-node) structure. The **vnop_readlink** entry point holds this v-node for the duration of the routine.
uio	Points to a **uio** structure. This structure contains the information required to read the link. In addition, it contains the return buffer for the **vnop_readlink** entry point.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_readlink** entry point is used by the logical file system to get the contents of a symbolic link, if the file system supports symbolic links. The logical file system finds the v-node (virtual node) for the symbolic link, so this routine simply reads the data blocks for the symbol link.

Execution Environment

The **vnop_readlink** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related reference:

"uio Structure" on page 638

Related information:

Virtual File System Kernel Extensions Overview

Logical File System Overview

vnop_rele Entry Point

Purpose

Releases a reference to a virtual node (v-node).

Syntax

```
int vnop_rele ( vp,)
struct vnode *vp;
```

Parameter

Item	Description
vp	Points to the v-node.

Description

The **vnop_rele** entry point is used by the logical file system to release the object associated with a v-node. If the object was the last reference to the v-node, the **vnop_rele** entry point then calls the **vn_free** kernel service to deallocate the v-node.

If the virtual file system (VFS) was unmounted while there were open files, the logical file system sets the **VFS_UNMOUNTING** flag in the **vfs** structure. If the flag is set and the v-node to be released is the last v-node on the chain of the **vfs** structure, then the virtual file system must be deallocated with the **vnop_rele** entry point.

Execution Environment

The **vnop_rele** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related reference:

“vn_free Kernel Service” on page 579

Related information:

Virtual File System Overview

Logical File System Overview

vnop_remove Entry Point

Purpose

Unlinks a file or directory.

Syntax

```
int vnop_remove ( vp, dvp, name, crp)
struct vnode *vp;
struct vnode *dvp;
char *name;
struct ucred *crp;
```

Parameters

Item	Description
vp	Points to a virtual node (v-node). The v-node indicates which file to remove and is held over the duration of the **vnop_remove** entry point.
dvp	Points to the v-node of the parent directory. This directory contains the file to be removed. The directory's v-node is held for the duration of the **vnop_remove** entry point.
name	Identifies the name of the file.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_remove** entry point is called by the logical file system to remove a directory entry (or link) as the result of a call to the **unlink** subroutine.

The logical file system assumes that the **vnop_remove** entry point calls the **vnop_rele** entry point. If the link is the last reference to the file in the file system, the disk resources that the file is using are released.

The logical file system ensures that the directory specified by the *dvp* parameter does not reside in a read-only file system.

Execution Environment

The **vnop_remove** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related reference:

"vnop_rele Entry Point" on page 680

Related information:

unlink subroutine

Virtual File System Overview

vnop_rename Entry Point

Purpose

Renames a file or directory.

Syntax

```
int vnop_rename (srcvp, srcdvp, oldname, destvp, destdvp, newname, crp)
struct vnode * srcvp;
struct vnode * srcdvp;
char * oldname;
struct vnode * destvp;
struct vnode * destdvp;
char * newname;
struct ucred * crp;
```

Parameters

Item	Description
srcvp	Points to the virtual node (v-node) of the object to rename.
srcdvp	Points to the v-node of the directory where the *srcvp* parameter resides. The parent directory for the old and new object can be the same.
oldname	Identifies the old name of the object.
destvp	Points to the v-node of the new object. This pointer is used only if the new object exists. Otherwise, this parameter is the null character.
destdvp	Points to the parent directory of the new object. The parent directory for the new and old objects can be the same.
newname	Points to the new name of the object.
crp	Points to the **cred** structure. This structure contains data that applications can use to validate access permission.

Description

The **vnop_rename** entry point is invoked by the logical file system to rename a file or directory. This entry point provides the following renaming actions:

- Renames an old object to a new object that exists in a different parent directory.
- Renames an old object to a new object that does not exist in a different parent directory.
- Renames an old object to a new object that exists in the same parent directory.
- Renames an old object to a new object that does not exist in the same parent directory.

To ensure that this entry point routine executes correctly, the logical file system guarantees the following:

- File names are not renamed across file systems.
- The old and new objects (if specified) are not the same.
- The old and new parent directories are of the same type of v-node.

Execution Environment

The **vnop_rename** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

rename subroutine

Virtual File System Overview

Logical File System Overview

vnop_revoke Entry Point

Purpose

Revokes all access to an object.

Syntax

```
int vnop_revoke (vp, cmd, flag, vinfop, crp)
struct vnode * vp;
int  cmd;
int  flag;
caddr_t  vinfop;
struct ucred * crp;
```

Parameters

Item	Description
vp	Points to the virtual node (v-node) containing the object.
cmd	Indicates whether the calling process holds the file open. This parameter takes the following values: **0** The process did not have the file open. **1** The process had the file open. **2** The process had the file open and the reference count in the file structure was greater than 1.
flag	Identifies the flags from the **file** structure.
vinfop	This parameter is currently unused.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_revoke** entry point is called to revoke further access to an object.

Execution Environment

The **vnop_revoke** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

frevoke subroutine

revoke subroutine

Virtual File System Overview

vnop_rmdir Entry Point

Purpose

Removes a directory.

Syntax

```
int vnop_rmdir ( vp,  dp,  pname,  crp)
struct vnode *vp;
struct vnode *dp;
char *pname;
struct ucred *crp;
```

Parameters

Item	Description
vp	Points to the virtual node (v-node) of the directory.
dp	Points to the parent of the directory to remove.
pname	Points to the name of the directory to remove.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_rmdir** entry point is invoked by the logical file system to remove a directory object. To remove a directory, the directory must be empty (except for the current and parent directories). Before removing the directory, the logical file system ensures the following:

- The *vp* parameter is a directory.
- The *vp* parameter is not the root of a virtual file system.
- The *vp* parameter is not the current directory.
- The *dp* parameter does not reside on a read-only file system.

Note: The *vp* and *dp* parameters' v-nodes (virtual nodes) are held for the duration of the routine.

Execution Environment

The **vnop_rmdir** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

rmdir subroutine

Virtual File System Overview

Logical File System Overview

vnop_seek Entry Point

Purpose

Validates file offsets.

Syntax

```
int vnop_seek (vp, offsetp, crp)
struct vnode * vp;
offset_t * offp;
struct ucred * crp;
```

Parameters

Item	Description
vp	Points to the virtual node (vnode) of the file.
offp	Points to the location of the new offset to validate.
crp	Points to the user's credential.

Description

The **vnop_seek** entry point is called by the logical file system to validate a new offset that is computed by the **lseek**, **llseek**, and **lseek64** subroutines. The file system implementation must check the offset that is pointed to by the *offp* parameter and, if it is acceptable for the file, return zero. If the offset is not acceptable, the routine must return a non-zero value. **EINVAL** is the suggested error value for invalid offsets.

File system that do not want to do offset validation can simply return 0. File system that do not provide the **vnop_seek** entry point has a maximum offset of **OFF_MAX** (2 gigabytes minus 1) enforced by the logical file system.

Execution Environment

The **vnop_seek** *entry* point is to be called from the process environment only.

Return Values

Item	Description
0	Indicates success.
Non-zero	Return values are returned the **/usr/include/sys/errno.h** file to indicate failure.

Related information:
lseek, llseek, and, lseek64

vnop_select Entry Point

Purpose

Polls a virtual node (v-node) for immediate I/O.

Syntax

```
int vnop_select (vp, correl, e, re, notify, vinfo, crp)
struct vnode * vp;
int  correl;
int  e;
int  re;
int (* notify)();
caddr_t  vinfo;
struct ucred * crp;
```

Parameters

Item	Description
vp	Points to the v-node to be polled.
correl	Specifies the ID used for correlation in the **selnotify** kernel service.
e	Identifies the requested event.
re	Returns an events list. If the v-node is ready for immediate I/O, this field should be set to indicate the requested event is ready.
notify	Specifies the subroutine to call when the event occurs. This parameter is for nested polls.
vinfo	Is currently unused.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Description

The **vnop_select** entry point is invoked by the logical file system to poll a v-node to determine if it is immediately ready for I/O. This entry point is used to implement the **select** and **poll** subroutines.

File system implementation can support constructs, such as devices or pipes, that support the select semantics. The **fp_select** kernel service provides more information about select and poll requests.

Execution Environment

The **vnop_select** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related reference:

"fp_select Kernel Service" on page 166

Related information:

select subroutine

Virtual File System Kernel Extensions Overview

vnop_setacl Entry Point

Purpose

Sets the access control list (ACL) for a file.

Syntax

```
#include <sys/acl.h>

int vnop_setacl ( vp,  uiop,  crp)
struct vnode *vp;
struct uio *uiop;
struct ucred *crp;
```

Description

The **vnop_setacl** entry point is used by the logical file system to set the access control list (ACL) on a file.

Parameters

Item	Description
vp	Specifies the virtual node (v-node) of the file system object.
uiop	Specifies the **uio** structure that defines the storage for the call arguments.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Execution Environment

The **vnop_setacl** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure. Valid values include:

Item	Description
ENOSPC	Indicates that the space cannot be allocated to hold the new ACL information.
EPERM	Indicates that the effective user ID of the process is not the owner of the file and the process is not privileged.

Related information:

chacl subroutine

statacl subroutine

Virtual File System Overview

vnop_setattr Entry Point

Purpose

Sets attributes of a file.

Syntax

```
int vnop_setattr (vp, cmd, arg1, arg2, arg3, crp)
struct vnode * vp;
int  cmd;
int  arg1;
int  arg2;
int  arg3;
struct ucred * crp;
```

Description

The **vnop_setattr** entry point is used by the logical file system to set the attributes of a file. This entry point is used to implement the **chmod**, **chownx**, and **utime** subroutines.

The values that the *arg* parameters take depend on the value of the *cmd* parameter. The **vnop_setattr** entry point accepts the following *cmd* values and *arg* parameters:

Possible cmd Values for the vnop_setattr Entry Point

Command	**V_OWN**	**V_UTIME**	**V_MODE**
arg1	**int** *flag*;	**int** *flag*;	**int** *mode*;
arg2	**int** *uid*;	**timestruc_t** **atime*;	Unused
arg3	**int** *gid*;	**timestruc_t** **mtime*;	Unused

Note: For **V_UTIME**, if arg2 or arg3 is NULL, then the corresponding time field, *atime* and *mtime*, of the file should be left unchanged.

Parameters

Item	**Description**
vp	Points to the virtual node (v-node) of the file.
cmd	Defines the setting operation. This parameter takes the following values:

V_OWN
Sets the user ID (UID) and group ID (GID) to the UID and GID values of the new file owner. The *flag* argument indicates which ID is affected.

V_UTIME
Sets the access and modification time for the new file. If the *flag* parameter has the value of **T_SETTIME**, then the specific values have not been provided and the access and modification times of the object should be set to current system time. If the **T_SETTIME** value is not specified, the values are specified by the *atime* and *mtime* variables.

V_MODE
Sets the file mode.

The **/usr/include/sys/vattr.h** file contains the definitions for the three command values.

arg1, arg2, arg3	Specify the command arguments. The values of the command arguments depend on which command calls the **vnop_setattr** entry point.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Execution Environment

The **vnop_setattr** entry point can be called from the process environment only.

Return Values

Item	**Description**
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

chmod subroutine

Understanding Virtual Nodes (V-nodes)

vnop_setxacl Entry Point

Purpose

Sets the access control list (ACL) for a file system object. This is an advanced interface compared to **vnop_setacl** and provides for ACL-type-based operations.

Syntax

```
#include <sys/acl.h>
int vnop_setxacl (vp, ctl_flags, acl_type, uiop, mode_info, crp)

struct vnode   *vp;
uint64_t       ctl_flags;
acl_type_t     acl_type;
struct uio     *uiop;
mode_t          mode_info;
struct ucred   *crp;
```

Description

The **vnop_setxacl** entry point sets the access control list (ACL) on a file. It is an advanced version of **vnop_setacl** interface and provides for ACL-type-based operations. This interface can also be used to manage special bits in mode word (such as SUID, SGID and SVTX) in case the ACL type does not support these bits through ACL.

Parameters

Item	Description
vp	Specifies the virtual node (v-node) of the file system object for which the ACL needs to be set.
acl_type	Specifies the ACL type of the ACL information that needs to be set for the file system object. **Note:** If the underlying physical file system does not support the ACL type being requested, the system could return an error.
acl_len	Pointer to a *length* variable. The space pointed to is used as an input, as well as output, parameter. As input, the value will indicate the size of buffer *uiop*. When the call returns, this space holds the actual length of the ACL (true for when the call is successful or when the call fails with **errno** set to **ENOSPC**).
ctl_flags	This 64-bit bit mask provides for control over the ACL setting and for any future variations in the interface. The following flag values have been defined: **SET_MODE_S_BITS** Indicates that the *mode_info* value is set by the caller and the ACL put operation must consider this value to complete the ACL put operation. **SET_ACL** Indicates that the ACL arguments point to valid ACL data that must be considered while the ACL put operation is being performed. **Note:** Both of the preceding values can be specified by the caller by ORing the two masks.
uiop	Specifies the **uio** structure that defines the storage for the call arguments.
mode_info	This value indicates any mode word information that needs to be set for the file system object as part of this ACL put operation. When mode bits are altered by specifying the **SET_MODE_S_BITS** flag (in *ctl_flags*), the entire ACL put operation will fail if the caller does not have the required privileges.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Execution Environment

The **vnop_setxacl** entry point can be called from the process environment only.

Return Values

Upon successful completion, the **vnop_setxacl** entry point returns 0. Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Item	Description
EPERM	Indicates that the effective user ID of the process is not authorized to change the ACL on the specified file system object.
EINVAL	Invalid operation. File system might not support the ACL type being set.

Note: This list of error numbers is not complete and is dependent on the particular physical file system implementation supporting the ACL.

Related reference:

"vnop_setacl Entry Point" on page 686

Related information:

statacl subroutine

Logical File System Overview

vnop_strategy Entry Point

Purpose

Accesses blocks of a file.

Syntax

```
int vnop_strategy ( vp,  bp,  crp)
struct vnode *vp;
struct buf *bp;
struct ucred *crp;
```

Description

Note: The **vnop_strategy** entry point is not implemented in Version 3.2 of the operating system.

The **vnop_strategy** entry point accesses blocks of a file. This entry point is intended to provide a block-oriented interface for servers for efficiency in paging.

Parameters

Item	Description
vp	Points to the virtual node (v-node) of the file.
bp	Points to a **buf** structure that describes the buffer.
crp	Points to the **cred** structure. This structure contains data that applications can use to validate access permission.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related reference:

"buf Structure" on page 614

Related information:

Virtual File System Overview

Virtual File System Kernel Extensions Overview

vnop_symlink Entry Point

Purpose

Creates a symbolic link.

Syntax

```
int vnop_symlink ( vp,  linkname,  target,  crp)
struct vnode *vp;
char *linkname;
char *target;
struct ucred *crp;
```

Description

The **vnop_symlink** entry point is called by the logical file system to create a symbolic link. The path name specified by the *linkname* parameter is the name of the new symbolic link. This symbolic link points to the object named by the *target* parameter.

Parameters

Item	Description
vp	Points to the virtual node (v-node) of the parent directory where the link is created.
linkname	Points to the name of the new symbolic link. The logical file system guarantees that the new link does not already exit.
target	Points to the name of the object to which the symbolic link points. This name need not be a fully qualified path name or even an existing object.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Execution Environment

The **vnop_symlink** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

symlink subroutine

Virtual File System Overview

Logical File System Overview

vnop_unmap Entry Point

Purpose

Unmaps a file.

Syntax

```
int vnop_unmap ( vp,  flag,  crp)
struct vnode *vp;
ulong flag;
struct ucred *crp;
```

Description

The **vnop_unmap** entry point is called by the logical file system to unmap a file. When this entry point routine completes successfully, the use count for the memory object should be decremented and (if the use count went to 0) the memory object should be destroyed. The file system implementation is required to perform only those operations that are unique to the file system. The logical file system handles virtual-memory management operations.

Parameters

Item	Description
vp	Points to the v-node (virtual node) of the file.
flag	Indicates how the file was mapped. This flag takes the following values: **SHM_RDONLY** The virtual memory object is read-only. **SHM_COPY** The virtual memory object is copy-on-write.
crp	Points to the **cred** structure. This structure contains data that the file system can use to validate access permission.

Execution Environment

The **vnop_unmap** entry point can be called from the process environment only.

Return Values

Item	Description
0	Indicates success.

Nonzero return values are returned from the **/usr/include/sys/errno.h** file to indicate failure.

Related information:

Virtual File System Overview

Virtual File System Kernel Extensions Overview

Logical File System Overview

Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year).

Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

Printed in USA