Bull

AIX 5L Technical Reference
Kernel and Subsystems

Volume 2/2

AIX

ORDER REFERENCE
86 A2 52EF 02

Bull

AIX 5L Technical Reference
Kernel and Subsystems

Volume 2/2

AIX

Software

May 2003

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 52EF 02

Printed in France

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

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

Trademarks and Acknowledgements

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

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

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

Linux is a registered trademark of Linus Torvalds.

Contents

About This Book

This book provides information about kernel services, device driver operations, file system operations, subroutines, the configuration subsystem, the communications subsystem, the low function terminal (LFT) subsystem, the logical volume subsystem, the M-audio capture and playback adapter subsystem, the printer subsystem, the SCSI subsystem, and the serial DASD subsystem.

This book is part of the six-volume technical reference set, *AIX 5L Version 5.2 Technical Reference*, that provides information on system calls, kernel extension calls, and subroutines in the following volumes:

- *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1* and *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2* provide information on system calls, subroutines, functions, macros, and statements associated with base operating system runtime services.
- *AIX 5L Version 5.2 Technical Reference: Communications Volume 1* and *AIX 5L Version 5.2 Technical Reference: Communications Volume 2* provide information on entry points, functions, system calls, subroutines, and operations related to communications services.
- *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1* and *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 2* provide information about kernel services, device driver operations, file system operations, subroutines, the configuration subsystem, the communications subsystem, the low function terminal (LFT) subsystem, the logical volume subsystem, the M-audio capture and playback adapter subsystem, the printer subsystem, the SCSI subsystem, and the serial DASD subsystem.

This edition supports the release of AIX 5L Version 5.2 with the 5200-01 Recommended Maintenance package. Any specific references to this maintenance package are indicated as *AIX 5.2 with 5200-01*.

Who Should Use This Book

This book is intended for system programmers wishing to extend the kernel. To use the book effectively, you should be familiar with operating system concepts and kernel programming.

Highlighting

The following highlighting conventions are used in this book:

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

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.

32-Bit and 64-Bit Support for the UNIX98 Specification

Beginning with Version 4.3, the operating system is designed to support The Open Group's UNIX98 Specification for portability of UNIX-based operating systems. Many new interfaces, and some current ones, have been added or enhanced to meet this specification, making Version 4.3 even more open and portable for applications.
At the same time, compatibility with previous releases of the operating system is preserved. This is accomplished by the creation of a new environment variable, which can be used to set the system environment on a per-system, per-user, or per-process basis.
To determine the proper way to develop a UNIX98-portable application, you may need to refer to The Open Group's UNIX98 Specification, which can be obtained on a CD-ROM by ordering *Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification*, a book which includes The Open Group's UNIX98 Specification on a CD-ROM.

Related Publications

The following books contain information about or related to application programming interfaces:

- *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*
- *AIX 5L Version 5.2 Communications Programming Concepts*
- *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*
- *AIX 5L Version 5.2 System Management Guide: Operating System and Devices*
- *AIX 5L Version 5.2 Files Reference*

Chapter 1. Configuration Subsystem

attrval Device Configuration Subroutine

Purpose

Verifies that attribute values are within range.

Syntax

```
#include <cf.h>
#include <sys/cfgodm.h>
#include <sys/cfgdb.h>

int attrval (uniquetype, pattr, errattr)
char *  uniquetype;
char *  pattr;
char **  errattr;
```

Parameters

uniquetype	Identifies the predefined device object, which is a pointer to a character string of the form `class/subclass/type`.
pattr	Points to a character string containing the attribute-value pairs to be validated, in the form `attr1=val1 attr2=val2`.
errattr	Points a pointer to a null-terminated character string. On return from the **attrval** subroutine, this string will contain the names of invalid attributes, if any are found. Each attribute name is separated by spaces.

Description

The **attrval** subroutine is used to validate each of a list of input attribute values against the legal range. If no illegal values are found, this subroutine returns a value of 0. Otherwise, it returns the number of incorrect attributes.

If any attribute values are invalid, a pointer to a string containing a list of invalid attribute names is returned in the *errattr* parameter. These attributes are separated by spaces.

Allocation of the error buffer is done in the **attrval** subroutine. However, a character pointer (for example, `char *errorb;`) must be declared in the calling routine. Thereafter, the address of that pointer is passed to the **attrval** subroutine (for example, `attrval(...,&errorb);`) as one of the parameters.

Return Values

0	Indicates that all values are valid.
Nonzero	Indicates the number of erroneous attributes.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

Related Information

List of Device Configuration Subroutines.

Predefined Attribute (PdAt) object class, Customized Attribute (CuAt) object class, Predefined Devices (PdDv) object class.

---

busresolve Device Configuration Subroutine

Purpose

Allocates bus resources for adapters on an I/O bus (including PCI, ISA, and Micro Channel adapters).

Syntax

```
#include <cf.h>
#include <sys/cfgodm.h>
#include <sys/cfgdb.h>

int busresolve
(logname, flag, conf_list,
not_res_list, busname)
char *  logname;
int   flags;
char *  conf_list;
char *   not_res_list;
char *  busname;
```

Parameters

logname	Specifies the device logical name.
flags	Specifies either the boot phase or 0.
conf_list	Points to an array of at least 512 characters.
not_res_list	Points to an array of at least 512 characters.
busname	Specifies the logical name of the bus.

Description

Note: Micro Channel and plug-in ISA adapters are only supported by AIX 5.1 or earlier.

The **busresolve** device configuration subroutine is invoked by a device's configuration method to allocate bus resources for all devices that have predefined bus resource attributes. It also is invoked by the bus Configuration method to resolve attributes of all devices in the Defined state.

This subroutine first queries the Customized Attribute and Predefined Attribute object classes to retrieve a list of current bus resource attribute settings and a list of possible settings for each attribute. To resolve conflicts between the values assigned to an already available device and the current device, the subroutine adjusts the values for attributes of devices in the Defined state. For example, the **busresolve** subroutine makes sure that the current device is not assigned the same interrupt level as an already available device when invoked at run time. These values are updated in the customized Attribute object class.

The **busresolve** subroutine never modifies attributes of devices that are already in the Available state. It ignores devices in the Defined state if their `chgstatus` field in the Customized Devices object class indicates that they are missing.

When the *logname* parameter is set to the logical name of a device, the **busresolve** subroutine adjusts the specified device's bus resource attributes if necessary to resolve any conflicts with devices that are already in the Available state. A device's Configuration method should invoke the **busresolve** subroutine to

ensure that its bus resources are allocated properly when configuring the device at run time. The Configuration method does not need to do this when run as part of system boot because the bus device's Configuration method would have already performed it.

If the *logname* parameter is set to a null string, the **busresolve** subroutine allocates bus resources for all devices that are not already in the Available state. The bus device's Configuration method invokes the **busresolve** subroutine in this way during system boot.

The *flags* parameter is set to 1 for system boot phase 1; 2 for system boot phase 2; and 0 when the **busresolve** subroutine is invoked during run time. The **busresolve** subroutine can be invoked only to resolve a specific device's bus resources at run time. That is, the *flags* parameter must be 0 when the *logname* parameter specifies a device logical name.

The **E_BUSRESOURCE** value indicates that the **busresolve** subroutine was not able to resolve all conflicts. In this case, the *conf_list* parameter will contain a list of the logical names of the devices for which it successfully resolved attributes. The *not_res_list* parameter will also contain a list of the logical names of the devices for which it could not successfully resolve all attributes. Devices whose names appear in the *not_res_list* parameter must not be configured into the Available state.

When writing a Configure method for a device having bus resources, make sure that it fails and returns a value of **E_BUSRESOURCE** if the **busresolve** subroutine does not return an **E_OK** value.

Note: If the *conf_list* and *not_res_list* strings are not at least 512 characters, there may be insufficient space to hold the device names.

Return Values

E_OK	Indicates that all bus resources were resolved and allocated successfully.
E_ARGS	Indicates that the parameters to the **busresolve** subroutine were not valid. For example, the *logname* parameter specifies a device logical name, but the *flags* parameter is not set to 0 for run time.
E_MALLOC	Indicates that the **malloc** operation if necessary memory storage failed.
E_NOCuDv	Indicates that there is no customized device data for the bus device whose logical name is specified by the *busname* parameter.
E_ODMGET	Indicates that an ODM error occurred while retrieving data from the Configuration data base.
E_PARENTSTATE	Indicates that the bus device whose name is specified by the *busname* parameter is not in the Available state.
E_BUSRESOLVE	Indicates that a bus resource for a device did not resolve. The *logname* parameter can identify the particular device. However, if this parameter is null, then an **E_BUSRESOLVE** value indicates that the bus resource for some unspecified device in the system did not resolve.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

ODM Device Configuration Object Classes.

List of Device Configuration Subroutines.

Related Information

Understanding ODM Object Classes and Objects in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

genmajor Device Configuration Subroutine

Purpose

Generates the next available major number for a device driver instance.

Syntax

```
#include <cf.h>
#include <sys/cfgodm.h>
#include <sys/cfgdb.h>

long genmajor ( device_driver_instance_name)
char *device_driver_instance_name;
```

Parameters

device_driver_instance_name	Points to a character string containing the device driver instance name.

Description

The **genmajor** device configuration subroutine is one of the routines designated for accessing the Customized Device Driver (CuDvDr) object class. If a major number already exists for the given device driver instance, it is returned. Otherwise, a new major number is generated.

The **genmajor** subroutine creates an entry (object) in the CuDvDr object class for the major number information. The lowest available major number or the major number that has already been allocated is returned. The CuDvDr object class is locked exclusively by this routine until its completion.

Return Values

If the **genmajor** subroutine executes successfully, a major number is returned. This major number is either the lowest available major number or the major number that has already been allocated to the device instance.

A value of -1 is returned if the **genmajor** subroutine fails.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

Related Information

The **reldevno** device configuration subroutine, **relmajor** device configuration subroutine.

List of ODM Commands and Subroutines in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

Customized Device Driver (CuDvDr) object class.

List of Device Configuration Subroutines.

genminor Device Configuration Subroutine

Purpose

Generates either the smallest unused minor number available for a device, a preferred minor number if it is available, or a set of unused minor numbers for a device.

Syntax

```
#include <cf.h>
#include <sys/cfgodm.h>
#include <sys/cfgdb.h>

long *genminor (device_instance, major_no, preferred_minor,
     minors_in_grp, inc_within_grp, inc_btwn_grp)
char * device_instance;
long  major_no;
int  preferred_minor;
int  minors_in_grp;
int  inc_within_grp;
int  inc_btwn_grp;
```

Parameters

device_instance	Points to a character string containing the device instance name.
major_no	Contains the major number of the device instance.
preferred_minor	Contains a single preferred minor number or a starting minor number for generating a set of numbers. In the latter case, the **genminor** subroutine can be used to get a set of minor numbers in a single call.
minors_in_grp	Indicates how many minor numbers are to be allocated.
inc_within_grp	Indicates the interval between minor numbers.
inc_btwn_grp	Indicates the interval between groups of minor numbers.

Description

The **genminor** device configuration subroutine is one of the designated routines for accessing the Customized Device Driver (CuDv) object class. To ensure that unique numbers are generated, the object class is locked by this routine until its completion.

If a single preferred minor number needs to be allocated, it should be given in the *preferred_minor* parameter. In this case, the other parameters should contain an integer value of 1. If the desired number is available, it is returned. Otherwise, a null pointer is returned, indicating that the requested number is in use.

If the caller has no preference and only requires one minor number, this should be indicated by passing a value of -1 in the *preferred_minor* parameter. The other parameters should all contain the integer value of 1. In this case, the **genminor** subroutine returns the lowest available minor number.

If a set of numbers is desired, then every number in the designated set must be available. An unavailable number is one that has already been assigned. To get a specific set of minor numbers allocated, the *preferred_minor* parameter contains the starting minor number. If this set has a minor number that is unavailable, then the **genminor** subroutine returns a null pointer indicating failure.

If the set of minor numbers needs to be allocated with the first number beginning on a particular boundary (that is, a set beginning on a multiple of 8), then a value of -1 should be passed in the *preferred_minor*

parameter. The *inc_btwn_grp* parameter should be set to the multiple desired. The **genminor** subroutine uses the *inc_btwn_grp* parameter to find the first complete set of available minor numbers.

If a list of minor numbers is to be returned, the return value points to the first in a list of preferred minor numbers. This pointer can then be incremented to move through the list to access each minor number. The minor numbers are returned in ascending sorted order.

Return Values

In the case of failure, a null pointer is returned. If the **genminor** subroutine succeeds, a pointer is returned to the lowest available minor number or to a list of minor numbers.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

Related Information

The **genmajor** device configuration subroutine, **getminor** device configuration subroutine, **reldevno** device configuration subroutine.

List of ODM Commands and Subroutines in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

Customized Device Driver (CuDvDr) object class.

List of Device Configuration Subroutines.

---

genseq Device Configuration Subroutine

Purpose

Generates a unique sequence number for creating a device's logical name.

Syntax

```
#include <cf.h>
#include <sys/cfgodm.h>
#include <sys/cfgdb.h>
int genseq (prefix)
char *prefix;
```

Parameters

prefix	Points to the character string containing the prefix name of the device.

Description

The **genseq** device configuration subroutine generates a unique sequence number to be concatenated with the device's prefix name. The device name in the Customized Devices (CuDv) object class is the concatenation of the prefix name and the sequence number. The rules for generating sequence numbers are as follows:

- A sequence number is a nonnegative integer. The smallest sequence number is 0.

- When deriving a device instance logical name, the next available sequence number (relative to a given prefix name) is allocated. This next available sequence number is defined to be the smallest sequence number not yet allocated to device instances using the same prefix name.
- Whether a sequence number is allocated or not is determined by the device instances in the CuDv object class. If an entry using the desired prefix exists in this class, then the sequence number for that entry has already been allocated.

It is up to the application to convert this sequence number to character format so that it can be concatenated to the prefix to form the device name.

Return Values

If the **genseq** subroutine succeeds, it returns the generated sequence number in integer format. If the subroutine fails, it returns a value of -1.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

Related Information

Customized Devices (CuDv) object class.

List of ODM Commands and Subroutines in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

---

getattr Device Configuration Subroutine

Purpose

Returns current values of an attribute object.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <cf.h>
#include <sys/cfgodm.h>
#include <sys/cfgdb.h>

struct CuAt *getattr (devname, attrname, getall, how_many)
char *  devname;
char *  attrname;
int  getall;
int *  how_many;
```

Parameters

devname	Specifies the device logical name.
attrname	Specifies the attribute name.
getall	Specifies a Boolean flag that, when set to True, indicates that a list of attributes is to be returned to the calling routine.
how_many	Points to how many attributes the **getattr** subroutine has found.

Description

The **getattr** device configuration subroutine returns the current value of an attribute object or a list of current values of attribute objects from either the Customized Attribute (CuAt) object class or the Predefined Attribute (PdAt) object class. The **getattr** device configuration subroutine queries the CuAt object class for the attribute object matching the device logical name and the attribute name. It is the application's responsibility to lock the Device Configuration object classes.

The **getattr** subroutine allocates memory for CuAt object class structures that are returned. This memory is automatically freed when the application exits. However, the application must free this memory if it invokes **getattr** several times and runs for a long time.

To get a single attribute, the *getall* parameter should be set to False. If the object exists in the CuAt object class, a pointer to this structure is returned to the calling routine.

However, if the object is not found, the **getattr** subroutine assumes that the attribute takes the default value found in the PdAt object class. In this case, the PdAt object class is queried for the attribute information. If this information is found, the relevant attribute values (that is, default value, flag information, and the NLS index) are copied into a CuAt structure. This structure is then returned to the calling routine. Otherwise, a null pointer is returned indicating an error.

To get all the customized attributes for the device name, the *getall* parameter should be set to True. In this case, the *attrname* parameter is ignored. The PdAt and CuAt object classes are queried and a list of CuAt structures is returned. The PdAt objects are copied to CuAt structures so that one list may be returned.

Note: The **getattr** device configuration subroutine will fail unless you first call the **odm_initialize** subroutine.

Return Values

Upon successful completion, the **getattr** subroutine returns a pointer to a list of CuAt structures. If the operation is unsuccessful, a null pointer is returned.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

Related Information

The **odm_initialize** subroutine, the **putattr** device configuration subroutine.

Predefined Attribute (PdAt) object class, Customized Attribute (CuAt) object class.

Device Configuration Subsystem Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding ODM Object Classes and Objects in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

ODM Device Configuration Object Classes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

getminor Device Configuration Subroutine

Purpose

Gets the minor numbers associated with a major number from the Customized Device Driver (CuDvDr) object class.

Syntax

```
#include <cf.h>
#include <sys/cfgodm.h>
#include <sys/cfgdb.h>

long *getminor (major_no, how_many, device_instance)
long   major_no;
int *  how_many;
char *  device_instance;
```

Parameters

major_no	Specifies the major number for which the corresponding minor number or numbers is desired.
how_many	Points to the number of minor numbers found corresponding to the *major_no* parameter.
device_instance	Specifies a device instance name to use when searching for minor numbers. This parameter is used in conjunction with the *major_no* parameter.

Description

The **getminor** device configuration subroutine is one of the designated routines for accessing the CuDvDr object class. This subroutine queries the CuDvDr object class for the minor numbers associated with the given major number or device instance or both.

If the *device_instance* parameter is null, then only the *major_no* parameter is used to obtain the minor numbers. Otherwise, both the *major_no* and *device_instance* parameters should be used. The number of minor numbers found in the query is returned in the *how_many* parameter.

The CuDvDr object class is locked exclusively by the **getminor** subroutine for the duration of the routine.

The return value pointer points to a list that contains the minor numbers associated with the major number. This pointer is then used to move through the list to access each minor number. The minor numbers are returned in ascending sorted order.

The **getminor** subroutine also returns the number of minor numbers in the list to the calling routine in the *how_many* parameter.

Return Values

If the **getminor** routine fails, a null pointer is returned.

If the **getminor** subroutine succeeds, one of two possible values is returned. If no minor numbers are found, null is returned. In this case, the *how_many* parameter points to an integer value of 0. However, if minor numbers are found, then a pointer to a list of minor numbers is returned. The minor numbers are returned in ascending sorted order. In the latter case, the *how_many* parameter points to the number of minor numbers found.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

Related Information

The **genminor** device configuration subroutine, **genmajor** device configuration subroutine, **reldevno** device configuration subroutine.

Customized Device Driver (CuDvDr) object class.

List of Device Configuration Subroutines.

---

loadext Device Configuration Subroutine

Purpose

Loads or unloads kernel extensions, or queries for kernel extensions in the kernel.

Syntax

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

mid_t loadext ( dd_name,  load,  query)
char *dd_name;
int load, query;
```

Parameters

dd_name	Specifies the name of the kernel extension to be loaded, unloaded, or queried.
load	Specifies whether the **loadext** subroutine should load the kernel extension.
query	Specifies whether a query of the kernel extension should be performed.

Description

The **loadext** device configuration subroutine provides the capability to load or unload kernel extensions. It can also be used to obtain the kernel module identifier (kmid) of a previously loaded object file. The kernel extension name passed in the *dd_name* parameter is either the base name of the object file or contains directory path information. If the kernel extension path name supplied in the *dd_name* parameter has no leading **./** (dot, slash), **../** double-dot, slash), or **/** (slash) characters, then the **loadext** subroutine concatenates the **/usr/lib/drivers** file and the base name passed in the *dd_name* parameter to arrive at an absolute path name. Otherwise, the path name provided in the *dd_name* parameter is used unmodified.

If the *load* parameter has a value of True, the specified kernel extension and its **kmid** are loaded. If the specified object file has already been loaded into the kernel, its load count is incremented and a new copy is not loaded.

If the *load* parameter has a value of False, the action taken depends on the value of the *query* parameter. If *query* is False, the **loadext** routine requests an unload of the specified kernel extension. This causes the kernel to decrement the load count associated with the object file. If the load count and use count of the object file become 0, the kernel unloads the object file. If the *query* parameter is True, then the **loadext** subroutine queries the kernel for the kmid of the specified object file. This kmid is then returned to the caller.

If both the *load* and *query* parameters have a value of True, the load function is performed.

Attention: Repeated loading and unloading of kernel extensions may cause a memory leak.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

Return Values

Upon successful completion, the **loadext** subroutine returns the kmid. If an error occurs or if the queried object file is not loaded, the routine returns a null value.

Related Information

The **sysconfig** subroutine.

Understanding Kernel Extension Binding in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

putattr Device Configuration Subroutine

Purpose

Updates, deletes, or creates an attribute object in the Customized Attribute (CuAt) object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <cf.h>
#include <sys/cfgodm.h>
#include <sys/cfgdb.h>

int putattr ( cuobj)
struct CuAt *cuobj;
```

Parameters

cuobj	Specifies the attribute object.

Description

The **putattr** device configuration subroutine either updates an old attribute object, creates a new object for the attribute information, or deletes an existing object in the CuAt object class. The **putattr** subroutine queries the CuAt object class to determine whether an object already exists with the device name and attribute name specified by the *cuobj* parameter.

If the attribute is found in the CuAt object class and its value (as given in the *cuobj* parameter) is to be changed back to the default value for this attribute, the customized object is deleted. Otherwise, the customized object is simply updated.

If the attribute object does not already exist and its attribute value is being changed to a non-default value, a new object is added to the CuAt object class with the information given in the *cuobj* parameter.

Note: The **putattr** device configuration subroutine will fail unless you first call the **odm_initialize** subroutine.

Return Values

0	Indicates a successful operation.
-1	Indicates a failed operation.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

Related Information

The **odm_initialize** subroutine, the **getattr** device configuration subroutine.

Customized Attribute (CuAt) object class.

Device Configuration Subsystem Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Object Data Manager (ODM) Overview for Programmers in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

---

reldevno Device Configuration Subroutine

Purpose

Releases the minor or major number, or both, for a device instance.

Syntax

#include <cf.h>
#include <sys/cfgodm.h>
#include <sys/cfgdb.h>
int reldevno (*device_instance_name***,** *release***)**
char ****device_instance_name*;**
int *release***;**

Parameters

device_instance_name	Points to the character string containing the device instance name.
release	Specifies whether the major number should be released. A value of True releases the major number; a value of False does not.

Description

The **reldevno** device configuration subroutine is one of the designated access routines to the Customized Device Driver (CuDvDr) object class. This object class is locked exclusively by this routine until its completion. All minor numbers associated with the device instance name are deleted from the CuDvDr object class. That is, each object is deleted from the class. This releases the minor numbers for reuse.

The major number is released for reuse if the following two conditions exist:

- The object to be deleted contains the last minor number for a major number.

- The *release* parameter is set to True.

If you prefer to release the major number yourself, then the **relmajor** device configuration subroutine can be called. In this case, you should also set the *release* parameter to False. All special files, including symbolically linked special files, corresponding to the deleted objects are deleted from the file system.

Return Values

0	Indicates successful completion.
-1	Indicates a failure to release the minor number or major number, or both.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

Related Information

The **genmajor** device configuration subroutine, **genminor** device configuration subroutine, **relmajor** device configuration subroutine.

Customized Device Driver (CuDvDr) object class.

---

relmajor Device Configuration Subroutine

Purpose

Releases the major number associated with the specified device driver instance name.

Syntax

#include <cf.h>
#include <sys/cfgodm.h>
#include <sys/cfgdb.h>
int relmajor (*device_driver_instance_name***)**
char ****device_driver_instance_name*;**

Parameter

device_driver_instance_name	Points to a character string containing the device driver instance name.

Description

The **relmajor** device configuration subroutine is one of the designated access routines to the Customized Device Driver (CuDvDr) object class. To ensure that unique major numbers are generated, the CuDvDr object class is locked exclusively by this routine until the major number has been released.

The **relmajor** routine deletes the object containing the major number of the device driver instance name.

Return Values

0	Indicates successful completion.
-1	Indicates a failure to release the major number.

Files

/usr/lib/libcfg.a	Archive of device configuration subroutines.

Related Information

The **genmajor** device configuration subroutine, **reldevno** device configuration subroutine.

Customized Device Driver (CuDvDr) object class.

Writing Optional Start and Stop Methods

This article describes how optional Start and Stop device methods work. It also suggests guidelines for programmers writing their own optional Start and Stop device configuration methods.

Syntax

stt*Dev* **-l** *Name*
stp*Dev* **-l** *Name*

Description

The Start and Stop methods are optional. They allow a device to support the additional device state of Stopped. The Start method takes the device from the Stopped state to the Available state. The Stop method takes the device from the Available state to the Stopped state. Most devices do not have Start and Stop methods.

The Stopped state keeps a configured device in the system, but renders it unusable by applications. In this state, the device's driver is loaded and the device is defined to the driver. This might be implemented by having the Stop method issue a command telling the device driver not to accept any normal I/O requests. If an application subsequently issues a normal I/O request to the device, it will fail. The Start method can then issue a command to the driver telling it to start accepting I/O requests once again.

If Start and Stop methods are written, the other device methods must be written to account for the Stopped state. For example, if a method checks for a device state of Available, it might now need to check for Available and Stopped states.

Additionally, write the Configure method so that it takes the device from the Defined state to the Stopped state. Also, the Configure method may invoke the Start method, taking the device to the Available state. The Unconfigure method must change the device to the Defined state from either the Available or Stopped states.

When used, Start and Stop methods are usually device-specific.

By convention, the first three characters of the name of the Start method are **stt**. The first three characters of the name of the Stop method are **stp**. The remainder of the names (*Dev*) can be any characters, subject to operating system file-name restrictions, that identify the device or group of devices that use the methods.

Flags

-l *name*	Identifies the logical name of the device to be started or stopped.

Related Information

Writing an Unconfigure Method , Writing a Configure Method .

Writing a Change Method

This article describes how a Change device method works. It also suggests guidelines for programmers writing their own Change device configuration methods.

Syntax

chg*Dev* **-l** *Name* [**-p** *Parent*] [**-w** *Connection*] [**-P** | **-T**] [**-a** *Attr*=*Value* [**-a** *Attr*=*Value* ...] ...]

Description

The Change method applies configuration changes to a device. If the device is in the Defined state, the changes are simply recorded in the Customized database. If the device is in the Available state, the Change method must also apply the changes to the actual device by reconfiguring it.

A Change method does not need to support all the flags described for Change methods. For example, if your device is a pseudo-device with no parent, it need not support parent and connection changes. For devices that have parents, it may be desirable to disallow parent and connection changes. For printers, such changes are logical because they are easily moved from one port to another. By contrast, an adapter card is not usually moved without first shutting off the system. It is then automatically configured at its new location when the system is rebooted. Consequently, there may not be a need for a Change method to support parent and connection changes.

Note: In deciding whether to support the **-T** and **-P** flags, remember that these options allow a device's configuration to get out of sync with the Configuration database. The **-P** flag is useful for devices that are typically kept open by the system. The Change methods for most supported devices do not support the **-T** flag.

In applying changes to a device in the Available state, the Change method could terminate the device from the driver, rebuild the device-dependent structure (DDS) using the new information, and redefine the device to the driver using the new DDS. The method may also need to reload adapter software or perform other device-specific operations. An alternative is to invoke the device's Unconfigure method, update the Customized database, and invoke the device's Configure method.

By convention, the first three characters of the name of the Change method should be **chg**. The remainder of the name (*Dev*) can be any characters, subject to operating system file-name restrictions, that identify the device or group of devices that use the method.

Flags

-l *Name*	Identifies the logical name of the device to be changed.
-p *Parent*	Identifies the logical name of a new parent for the device. This flag is used to move a device from one parent to another.
-w *Connection*	Identifies a new connection location for the device. This flag either identifies a new connection location on the device's existing parent, or if the **-p** flag is also used, it identifies the connection location on the new parent device.
-P	Indicates that the changes are to be recorded in the Customized database without those changes being applied to the actual device. This is a useful option for a device which is usually kept open by the system such that it cannot be changed. Changes made to the database with this flag are later applied to the device when it is configured at system reboot.

-T	Indicates that the changes are to be applied only to the actual device and not recorded in the database. This is a useful option for allowing temporary configuration changes that will not apply once the system is rebooted.
-a *Attr*=*Value*	Specifies the device attribute value pairs used for changing specific attribute values. The *Attr=Value* parameter contains one or more attribute value pairs for the **-a** flag. If you use a **-a** flag with multiple attribute value pairs, the list of pairs must be enclosed in quotes with spaces between the pairs. For example, entering **-a** *Attr=Value* lists one attribute value pair, while entering **-a** '*Attr1=Value1 Attr2=Value2*' lists more than one attribute value pair.

Guidelines for Writing a Change Method

This list of tasks is intended as a guideline for writing a Change method. When writing for a specific device, some tasks may be omitted. For example, if a device does not support the changing of a parent or connection, there is no need to include those tasks. A device may have special needs that are not included in these tasks.

If the Change method is written to invoke the Unconfigure and Configure methods, it must:

1. Validate the input parameters. The **-l** flag must be supplied to identify the device that is to be changed. If your method does not support the specified flag, exit with an error.
2. Initialize the Object Data Manager (ODM). Use the **odm_initialize** subroutine and lock the Configuration database using the **odm_lock** subroutine. See ″Writing a Define Method″ for an example.
3. Retrieve the Customized Device (CuDv) object for the device to be changed by getting the CuDv object whose Device Name descriptor matches the name supplied with the **-l** flag. If no object is found with the specified name, exit with an error.
4. Validate all attributes being changed. Make certain that the attributes apply to the specified device, that they can be set by the user, and that they are being set to valid values. The **attrval** subroutine can be used for this purpose. If some attributes have values that are dependent on each other, write the code to cross check them. If invalid attributes are found, the method needs to write information to standard error describing them. See ″Handling Invalid Attributes″ .
5. Determine if a new parent device exists. If a new parent device has been specified, find out whether it exists by querying the CuDv object class for an object whose Device Name descriptor matches the new parent name. If no match is found, the method exits with an error.
6. If a new connection has been specified, validate that this device can be connected there. Do this by querying the Predefined Connection (PdCn) object class for an object whose Unique Type descriptor matches the link to the Predefined Devices (PdDv) object class descriptor of the parent's CuDv object. The Connection Key descriptor of the CuDv object must match the subclass name of the device being changed, and the Connection Location descriptor of the CuDv object must match the new connection value. If no match is found, the method exits with an error.
7. If a match is found, the new connection is valid. If the device is in the Available state, then it should still be available after being moved to the new connection. Since only one device can be available at a particular connection, the Change method must check for other available devices at that connection. If one is found, the method exits with an error.
8. If the device state is Available and the **-P** flag was not specified, invoke the device's Unconfigure method using the **odm_run_method** command. This fails if the device has Available child devices, which is why the Change method does not need to check explicitly for child devices.
9. If any attribute settings were changed, update the database to reflect the new settings. If a parent or connection changed, update the Parent Device Logical Name, Location Where Connected on Parent Device, and Location Code descriptors of the device's CuDv object.
10. If the device state was in the Available state before being unconfigured, invoke the device's Configure method using the **odm_run_method** command. If this returns an error, leaving the device unconfigured, the Change method should restore the Customized database to its pre-change state.

11. Close all object classes and terminate the ODM. Exit with an exit code of 0 if there were no errors.

Handling Invalid Attributes

If the Change method detects attributes that are in error, it must write information to the **stderr** file to identify them. This consists of writing the attribute name followed by the attribute description. Only one attribute and its description is to be written per line. If an attribute name was mistyped so that it does not match any of the device's attributes, write the attribute name supplied on a line by itself.

The **mkdev** and **chdev** configuration commands intercept the information written to the standard error file by the Change method. These commands write out the information following an error message describing that there were invalid attributes. Both the attribute name and attribute description are needed to identify the attribute. By invoking the **mkdev** or **chdev** command directly, the attributes can be identified by name. When using SMIT, these attributes can be identified by description.

The attribute description is obtained from the appropriate message catalog. A message is identified by catalog name, set number, and message number. The catalog name and set number are obtained from the device's PdDv object. The message number is obtained from the NLS Index descriptor in either the Predefined Attribute (PdAt) or Customized Attribute (CuAt) object corresponding to the attribute.

Related Information

Writing an Unconfigure Method , Writing a Configure Method

The **chdev** command, **mkdev** command, **rmdev** command.

The **attrval** subroutine, **odm_run_method** subroutine.

Customized Devices (CuDv) object class, Predefined Devices (PdDv) object class, Predefined Connection (PdCn) object class, Predefined Attribute (PdAt) object class, Customized Attribute (CuAt) object class.

Device Dependent Structure (DDS) Overview, Understanding Device Dependencies and Child Devices in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Object Data Manager (ODM) Overview for Programmers in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

Writing a Configure Method

This article describes how a Configure device method works. It also suggests guidelines for programmers writing their own Configure device configuration methods.

Syntax

cfg*Dev* **-l** *Name* [**-1** | **-2**]

Description

The Configure method moves a device from Defined (not available for use in the system) to Available (available for use in the system). If the device has a driver, the Configure method loads the device driver into the kernel and describes the device characteristics to the driver. For an intermediate device (such as a SCSI bus adapter), this method determines which attached child devices are to be configured and writes their logical names to standard output.

The Configure method is invoked by either the **mkdev** configuration command or by the Configuration Manager. Because the Configuration Manager runs a second time in phase 2 system boot and can also be

invoked repeatedly at run time, a device's Configure method can be invoked to configure an Available device. This is not an error condition. In the case of an intermediate device, the Configure method checks for the presence of child devices. If the device is not an intermediate device, the method simply returns.

In general, the Configure method obtains all the information it needs about the device from the Configuration database. The options specifying the phase of system boot are used to limit certain functions to specific phases.

If the device has a parent device, the parent must be configured first. The Configure method for a device fails if the parent is not in the Available state.

By convention, the first three characters of the name of the Configure method are **cfg**. The remainder of the name (*Dev*) can be any characters, subject to operating system file-name restrictions, that identify the device or group of devices that use the method.

Flags

-l *Name*	Identifies the logical name of the device to be configured.
-1	Specifies that the device is being configured in phase 1 of the system boot. This option cannot be specified with the -2 flag. If neither the -1 nor the -2 flags are specified, the Configure method is invoked at runtime.
-2	Specifies that the device is being configured in phase 2 of the system boot. This option cannot be specified with the -1 flag. If neither the -1 nor the -2 flags are specified, the Configure method is invoked at runtime.

Handling Device Vital Product Data (VPD)

Devices that provide vital product data (VPD) are identified in the Predefined Device (PdDv) object class by setting the VPD flag descriptor to TRUE in each of the device's PdDv objects. The Configure method must obtain the VPD from the device and store it in the Customized VPD (CuVPD) object class. Consult the appropriate hardware documentation to determine how to retrieve the device's VPD. In many cases, VPD is obtained from the device driver using the **sysconfig** subroutine.

Once the VPD is obtained from the device, the Configure method queries the CuVPD object class to see if the device has hardware VPD stored there. If so, the method compares the VPD obtained from the device with that from the CuVPD object class. If the VPD is the same in both cases, no further processing is needed. If they are different, update the VPD in the CuVPD object class for the device. If there is no VPD in the CuVPD object class for the device, add the device's VPD.

By first comparing the device's VPD with that in the CuVPD object class, modifications to the CuVPD object class are reduced. This is because the VPD from a device typically does not change. Reducing the number of database writes increases performance and minimizes possible data loss.

Understanding Configure Method Errors

For many of the errors detected, the Configure method exits with the appropriate exit code. In other cases, the Configure method may need to undo some of the operations it has performed. For instance, after loading the device driver and defining the device to the driver, the Configure method may encounter an error while downloading microcode. If this happens, the method will terminate the device from the driver using the **sysconfig** subroutine and unload the driver using the **loadext** subroutine.

The Configure method does not delete the special files or unassign the major and minor numbers if they were successfully allocated and the special file created before the error was encountered. This is because the operating system's configuration scheme allows both major and minor numbers and special files to be maintained for a device even though the device is unconfigured.

If the device is configured again, the Configure method will recognize that the major and minor numbers are allocated and that the special files exist.

By the time the Configure method checks for child devices, it has successfully configured the device. Errors that occur while checking for child devices are indicated with the **E_FINDCHILD** exit code. The **mkdev** command detects whether the Configure method completed successfully. If needed, it will display a message indicating that an error occurred while looking for child devices.

Guidelines for Writing a Configure Method

The following tasks are guidelines for writing a Configure method. When writing for a specific device, some tasks may be omitted. For example, if the device is not an intermediate device or does not have a driver, the method is written accordingly. A device may also have special requirements not listed in these tasks.

The Configure method must:

1. Validate the input parameters. The **-l** logical name flag must be supplied to identify the device that is to be configured. The **-1** and **-2** flags cannot be supplied at the same time.
2. Initialize the Object Data Manager (ODM). Use the **odm_initialize** subroutine and lock the Configuration database using the **odm_lock** subroutine. See ″Writing a Define Method″ for an example.
3. Retrieve the Customized Device (CuDv) object for the device to be configured. The CuDv object's Device Name descriptor must match the name supplied with the **-l** logical name flag. If no object is found with the specified name, the method exits with an error.
4. Retrieve the Predefined Device (PdDv) object for the device to be configured. The PdDv object's Unique Type descriptor must match the link to PdDv object class descriptor of the device's CuDv object.
5. Obtain the LED value descriptor of the device's PdDv object. Retrieve the LED Value descriptor of the device's PdDv object and display this value on the system LEDs using the **setleds** subroutine if either the **-1** or **-2** flag is specified. This specifies when the Configure method will execute at boot time. If the system hangs during configuration at boot time, the displayed LED value indicates which Configure method created the problem.

   If the device is already configured (that is, the Device State descriptor of the device's CuDv object indicates the Available state) and is an intermediate device, skip to the task of detecting child devices. If the device is configured but is not an intermediate device, the Configure method will exit with no error.

   If the device is in the Defined state, the Configure Method must check the parent device, check for the presence of a device, obtain the device VPD, and update the device's CuDv object.
6. If the device has a parent, the Configure method validates the parent's existence and verifies that the parent is in the Available state. The method looks at the Parent Device Logical Name descriptor of the device's CuDv object to obtain the parent name. If the device does not have a parent, the descriptor will be a null string.

   When the device has a parent, the Configure method will obtain the parent device's CuDv object and check the Device State descriptor. If the object does not exist or is not in the Available state, the method exits with an error.

   Another check must be made if a parent device exists. The Configure method must verify that no other device connected to the same parent (at the same connection location) has been configured. For example, two printers can be connected to the same port using a switch box. While each printer has the same parent and connection, only one can be configured at a time.

   The Configure method performs this check by querying the CuDv object class. It queries for objects whose Device State descriptor is set to the Available state and whose Parent Device Logical Name and Location Where Connected on Parent Device descriptors match those for the device being configured. If a match is found, the method exits with an error.

7. Check the presence of the device. If the device is an adapter card and the Configure method has been invoked at run time (indicated by the absence of both the **-1** and **-2** flags), the Configure method must verify the adapter card's presence. This is accomplished by reading POS registers from the card. (The POS registers are obtained by opening and accessing the **/dev/bus0** or **/dev/bus1** special file.) This is essential, because if the card is present, the Configure method must invoke the **busresolve** library routine to assign bus resources to avoid conflict with other adapter cards in the system. If the card is not present or the **busresolve** routine fails to resolve bus resources, the method exits with an error.
8. Determine if the device has a device driver. The Configure method obtains the name of the device driver from the Device Driver Name descriptor of the device's PdDv object. If this descriptor is a null string, the device does not have a device driver.

   If the device has a device driver, the Configure method must:

   a. Load the device driver using the **loadext** subroutine. See ″Loading a Device Driver″ for more information.
   b. Determine the device's major number using the **genmajor** subroutine.
   c. Determine the device's minor number using the **getminor** or **genminor** subroutine or by your own device-dependent routine.
   d. Create special files in the **/dev** directory if they do not already exist. Special files are created with the **mknod** subroutine.
   e. Build the device-dependent structure (DDS). This structure contains information describing the characteristics of the device to the device driver. The information is usually, but not necessarily, obtained from the device's attributes in the Configuration database. Refer to the appropriate device driver information to determine what the device driver expects the DDS to look like. The ″Device Dependent Structure (DDS) Overview″ in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts* describes the DDS structure.
   f. Use the **sysconfig** subroutine to pass the DDS to the device driver.
   g. If code needs to be downloaded to the device, read in the required file and pass the code to the device through the interface provided by the device driver. The file to be downloaded might be identified by a Predefined Attribute (PdAt) or Customized Attribute (CuAt) object. By convention, microcode files are in the **/etc/microcode** directory (which is a symbolic link to the **/usr/lib/microcode** directory). Downloaded adapter software is in the **/usr/lib/asw** directory.
9. Obtain the device VPD. After the tasks relating to the device driver are complete, or if the device did not have a device driver, the Configure method will determine if it needs to obtain vital product data (VPD) from the device. The VPD Flag descriptor of the device's PdDv object specifies whether or not it has VPD. See ″Handling Device Vital Product Data (VPD)″ for more details.
10. Update the CuDv object. At this point, if no errors have been encountered, the device is configured. The Configure method will update the Device Status descriptor of the device's CuDv object to indicate that it is in the Available state. Also, set the Change Status descriptor to SAME if it is currently set to MISSING. This can occur if the device was not detected at system boot and is being configured at run time.
11. Define detected child devices not currently represented in the CuDv object class. To accomplish this, invoke the Define method for each new child device. For each detected child device already defined in the CuDv object class, the Configure method looks at the child device's CuDv Change Status Flag descriptor to see if it needs to be updated. If the descriptor's value is **DONT_CARE**, nothing needs to be done. If it has any other value, it must be set to SAME and the child device's CuDv object must be updated. The Change Status Flag descriptor is used by the system to indicate configuration changes.

    If the device is an intermediate device but cannot detect attached child devices, query the CuDv object class about this information. The value of the Change Status Flag descriptor for these child devices should be **DONT_CARE** because the parent device cannot detect them. Sometimes a child device has an attribute specifying to the Configure method whether the child device is to be configured. The **autoconfig** attribute of TTY devices is an example of this type of attribute.

Regardless of whether the child devices are detectable, the Configure method will write the device logical names of the child devices to be configured to standard output, separated by space characters. If the method was invoked by the Configuration Manager, the Manager invokes the Configure method for each of the child device names written to standard output.

12. Close all object classes and terminate the ODM. Close all object classes and terminate the ODM. If there are no errors, use a 0 (zero) code to exit.

Files

/dev/bus0	Contains POS registers.
/dev/bus1	Contains POS registers.
/etc/microcode directory	Contains microcode files. A symbolic link to the **/usr/lib/microcode** directory.
/usr/lib/asw directory	Contains downloaded adapter software.

Related Information

The **mkdev** command.

The **busresolve** subroutine, **genmajor** subroutine, **genminor** subroutine, **getminor** subroutine, **loadext** subroutine, **mknod** subroutine, **odm_initialize** subroutine, **odm_lock** subroutine, **reldevno** subroutine, **relmajor** subroutine, **sysconfig** subroutine.

Object Data Manager (ODM) Overview for Programmers in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

Customized Devices (CuDv) object class, Predefined Devices (PdDv) object class, Customized Attributes (CuAt) object class, Predefined Attribute (PdAt) object class, Customized Vital Product Data (CuVPD) object class.

Understanding Device States, Understanding Device Dependencies and Child Devices, Loading a Device Driver Configuration Manager Overview, Understanding System Boot Processing, Device Dependent Structure (DDS) Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Device Configuration Subsystem Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Special Files Overview in *AIX 5L Version 5.2 Files Reference*.

Writing a Device Method in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Writing an Unconfigure Method , Writing a Define Method .

Writing a Define Method

This article describes how a Define device method works. It also suggests guidelines for programmers writing their own Define device configuration methods.

Syntax

def*Dev* **-c** *Class* **-s** *SubClass* **-t** *Type* [**-p** *Parent* **-w** *Connection*] [**-l** *Name*]

Description

The Define method is responsible for creating a customized device in the Customized database. It does this by adding an object for the device into the Customized Devices (CuDv) object class. The Define method is invoked either by the **mkdev** configuration command, by a node configuration program, or by the Configure method of a device that is detecting and defining child devices.

The Define method uses information supplied as input, as well as information in the Predefined database, for filling in the CuDv object. If the method is written to support a single device, it can ignore the class, subclass, and type options. In contrast, if the method supports multiple devices, it may need to use these options to obtain the PdDv device object for the type of device being customized.

By convention, the first three characters of the name of the Define method should be **def**. The remainder of the name (*Dev*) can be any characters that identify the device or group of devices that use the method, subject to operating system file-name restrictions.

Flags

-c *Class*	Specifies the class of the device being defined. Class, subclass, and type are required to identify the Predefined Device object in the Predefined Device (PdDv) object class for which a customized device instance is to be created.
-s *SubClass*	Specifies the subclass of the device being defined. Class, subclass, and type are required to identify the Predefined Device object in the PdDv object class for which a customized device instance is to be created.
-t *Type*	Specifies the type of the device being defined. Class, subclass, and type are required to identify the predefined device object in the PdDv object class for which a customized device instance is to be created.
-p *Parent*	Specifies the logical name of the parent device. This logical name is required for devices that connect to a parent device. This option does not apply to devices that do not have parents; for example, most pseudo-devices.
-w *Connection*	Specifies where the device connects to the parent. This option applies only to devices that connect to a parent device.
-l *Name*	Passed by the **mkdev** command, specifies the name for the device if the user invoking the command is defining a new device and wants to select the name for the device. The Define method assigns this name as the logical name of the device in the Customized Devices (CuDv) object, if the name is not already in use. If this option is not specified, the Define method generates a name for the device. Not all devices support or need to support this option.

Guidelines for Writing a Define Method

This list of tasks is meant to serve as a guideline for writing a Define method. In writing a method for a specific device, some tasks may be omitted. For instance, if a device does not have a parent, there is no need to include all of the parent and connection validation tasks. Additionally, a device may have special needs that are not listed in these tasks.

The Define method must:

1. Validate the input parameters. Generally, a Configure method that invokes the child-device Define method is coded to pass the options expected by the child-device Define method. However, the **mkdev** command always passes the class, subclass, and type options, while only passing the other options based on user input to the **mkdev** command. Thus, the Define method may need to ensure that all of the options it requires have been supplied. For example, if the Define method expects parent and connection options for the device being defined, it must ensure that the options are supplied. Also, a Define method that does not support the **-l** name specification option will exit with an error if the option is supplied.

2. Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the configuration database using the **odm_lock** subroutine. The following code fragment illustrates this process:

```
#include <cf.h>

if (odm_initialize() < 0)
        exit(E_ODMINIT);         /* initialization failed */

if (odm_lock("/etc/objrepos/config_lock",0) == -1) {
        odm_terminate();
        exit(E_ODMLOCK);        /* database lock failed */
}
```

3. Retrieve the predefined PdDv object for the type of device being defined. This is done by obtaining the object from the PdDv object class whose class, subclass, and type descriptors match the class, subclass, and type options supplied to the Define method. If no match is found, the Define method will exit with an error. Information will be taken from the PdDv device object in order to create the CuDv device object.
4. Ensure that the parent device exists. If the device being defined connects to a parent device and the name of the parent has been supplied, the Define method must ensure that the specified device actually exists. It does this by retrieving the CuDv object whose Device Name descriptor matches the name of the parent device supplied using the **-p** flag. If no match is found, the Define method will exit with an error.
5. If the device has a parent and that parent device exists in the CuDv object class, validate that the device being defined can be connected to the specified parent device. To do this, retrieve the predefined connection object from the Predefined Connection (PdCn) object class whose Unique Type, Connection Key, and Connection Location descriptors match the Link to Predefined Devices Object Class descriptor of the parent's CuDv object obtained in the previous step and the subclass and connection options input into the Define method, respectively. If no match is found, an invalid connection is specified. This may occur because the specified parent is not an intermediate device, does not accept the type of device being defined (as described by subclass), or does not have the connection location identified by the connection option.
6. Assign a logical name to the device. Each newly assigned logical name must be unique to the system. If a name has been supplied using the **-l** flag, make certain it is unique before assigning it to the device. This is done by checking the CuDv object class for any object whose Device Name descriptor matches the desired name. If a match is found, the name is already used and the Define method must exit with an error.

   If the Define method is to generate a name, it can do so by obtaining the prefix name from the Prefix Name descriptor of the device's PdDv device object and invoking the **genseq** subroutine to obtain a unique sequence number for this prefix. Appending the sequence number to the prefix name results in a unique name. The **genseq** routine looks in the CuDv object class to ensure that it assigns a sequence number that has not been used with the specified prefix to form a device name.

   In some cases, a Define method may need to ensure that only one device of a particular type has been defined. For example, there can only be one pty device customized in the CuDv object class. The pty Define method does this by querying the CuDv object class to see if a device by the name `pty0` exists. If it does, the pty device has already been defined. Otherwise, the Define method proceeds to define the pty device using the name pty0.
7. Determine the device's location code. If the device being defined is a physical device, it has a location code. ″Location Codes″ in *AIX 5L Version 5.2 System Management Concepts: Operating System and Devices* has more information about location codes.
8. Create the new CuDv object.

   Set the CuDv object descriptors as follows:

Descriptor	Setting
Device name	Use the name as determined in step 6.
Device status flag	Set to the Defined state.

Descriptor	Setting
Change status flag	Set to the same value as that found in the Change Status Flag descriptor in the device's PdDv object.
Device driver instance	Set to the same value as the Device Driver Name descriptor in the device's PdDv object. This value may be used later by the Configure method.
Device location code	Set to a null string if the device does not have a location code. Otherwise, set it to the value computed.
Parent device logical name	Set to a null string if the device does not have a parent. Otherwise, set this descriptor to the parent name as specified by the parent option.
Location where connected on parent device	Set to a null string if the device does not have a parent. Otherwise, set this descriptor to the value specified by the connection option.
Link to predefined devices object class	Set to the value obtained from the Unique Type descriptor of the device's PdDv object.

9. Write the name of the device to standard output. A blank should be appended to the device name to serve as a separator in case other methods write device names to standard output. Either the **mkdev** command or the Configure method that invoked the Define method will intercept standard output to obtain the device name assigned to the device.
10. Close all object classes and terminate the ODM. Exit with an exit code of 0 if there were no errors.

Related Information

The **mkdev** command.

The **genseq** device configuration subroutine, **odm_initialize** subroutine, **odm_lock** subroutine.

Writing an Undefine Method , Writing a Configure Method .

Customized Devices (CuDv) object class, Predefined Devices object class, Predefined Connection object class, Predefined Attribute (PdAt) object class, Customized Attribute (CuAt) object class.

Understanding Device States, Understanding Device Classes, Subclasses, and Types, Understanding Device Dependencies and Child Devices, Loading A Device Driver Configuration Manager Overview, Understanding System Boot Processing, Writing a Device Method in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Object Data Manager (ODM) Overview for Programmers in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

Location Codes in *AIX 5L Version 5.2 System Management Concepts: Operating System and Devices*.

Writing an Unconfigure Method

This article describes how an Unconfigure device method works. It also suggests guidelines for programmers writing their own Unconfigure device configuration method.

Syntax

ucfg*Dev* **-l** *Name*

Description

The Unconfigure method takes an Available device (available for use in the system) to a Defined state (not available for use in the system). All the customized information about the device is retained in the database so that the device can be configured again exactly as it was before.

The actual operations required to make a device defined depend on how the Configure method made the device available in the first place. For example, if the device has a device driver, the Configure method must have loaded a device driver in the kernel and described the device to the driver through a device dependent structure (DDS). Then, the Unconfigure method must tell the driver to delete the device instance and request an unload of the driver.

If the device is an intermediate device, the Unconfigure method must check the states of the child devices. If any child device is in the Available state, the Unconfigure method fails and leaves the device configured. To ensure proper system operation, all child devices must be unconfigured before the parent can be unconfigured.

Although the Unconfigure method checks child devices, it does not check the device dependencies recorded in the Customized Dependency (CuDep) object class. See ″Understanding Device Dependencies and Child Devices″ in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

The Unconfigure method also fails if the device is currently open. In this case, the device driver returns a value for the **errno** global variable of **EBUSY** to the Unconfigure method when the method requests the driver to delete the device. The device driver is the only component at that instant that knows the device is open. As in the case of configured child devices, the Unconfigure method fails and leaves the device configured.

When requesting the device driver to terminate the device, the **errno** global variable values other than **EBUSY** can be returned. The driver should return **ENODEV** if it does not know about the device. Under the best circumstances, however, this case should not occur. If **ENODEV** is returned, the Unconfigure method should unconfigure the device so that the database and device driver are in agreement. If the device driver returns any other **errno** global value, it deletes any stored characteristics for the specified device instance. The Unconfigure method indicates that the device is unconfigured by setting the state to Defined.

The Unconfigure method does not generally release the major and minor number assignments for a device, or delete the device's special files in the **/dev** directory.

By convention, the first four characters of the name of the Unconfigure method should be **ucfg**. The remainder of the name (*Dev*) can be any characters, subject to operating system file-name restrictions, that identify the device or group of devices that use the method.

Flags

-l *Name* Identifies the logical name of the device to be unconfigured.

Guidelines for Writing an Unconfigure Method

This list of tasks is intended as a guideline for writing an Unconfigure method. When you write a method for a specific device, some tasks may be omitted. For example, if a device is not an intermediate device or does not have a driver, the method can be written accordingly. The device may have special needs that are not listed in these tasks.

The Unconfigure method must:

1. Validate the input parameters. The **-l** flag must be supplied to identify the device that is to be unconfigured.
2. Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the Configuration database using the **odm_lock** subroutine. See ″Writing a Define Method″ for an example.

3. Retrieve the customized device (CuDv) object for the device to be unconfigured. Use the CuDv object whose Device Name descriptor matches the name supplied with the **-l** flag. If no object is found with the specified name, the method exits with an error.
4. Check the state of the device. If the Device Status descriptor indicates that the device is in the Defined state, then it is already unconfigured. In this case, exit.
5. Check for child devices in the available state. This can be done by querying the CuDv object class for objects whose Parent Device Logical Name descriptor matches this device's name and whose Device Status descriptor is not Defined. If a match is found, this method must exit with an error.
6. Retrieve the Predefined Device (PdDv) object for the device to be unconfigured by getting the PdDv object whose Unique Type descriptor matches the Link to Predefined Devices Object Class descriptor of the device's CuDv object. This object will be used to get the device driver name.
7. Delete device instance from driver and unload driver. Determine if the device has a driver. The Unconfigure method obtains the name of the device from the Device Driver Name descriptor of the PdDv object. If this descriptor is a null string, the device does not have a driver. In this situation, skip to the task of updating the device's state.

   If the device has a device driver, the Unconfigure method needs to include the following tasks:

   a. Determine the device's major and minor numbers using the **genmajor** and **getminor** subroutines. These are used to compute the device's devno, using the **makedev** macro defined in the **/usr/include/sysmacros.h** file, in preparation for the next task.
   b. Use the **sysconfig** subroutine to tell the device driver to terminate the device. If a value of **EBUSY** for the **errno** global variable is returned, this method exits with an error.
   c. Use the **loadext** routine to unload the device driver from the kernel. The **loadext** subroutine will not actually unload the driver if there is another device still configured for the driver. See ″Loading a Device Driver″ for more details.
8. Set defined status. The device is now unconfigured. The Unconfigure method will update the Device Status descriptor of the device's CuDv object to the Defined state.
9. Close all object classes and terminate the ODM. If there are no errors, exit with an exit code of 0 (zero).

Files

/usr/include/sysmacros.h	Contains macro definitions.

Related Information

The **mkdev** command.

The **genmajor** subroutine, **getminor** subroutine, **loadext** subroutine, **odm_initialize** subroutine, **odm_lock** subroutine, **sysconfig** subroutine.

Writing a Configure Method , Loading A Device Driver , Writing a Define Method .

Customized Devices (CuDv) object class, Predefined Devices (PdDv) object class.

Object Data Manager (ODM) Overview for Programmers in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

The Device Dependent Structure (DDS) Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding Device States, Understanding Device Dependencies and Child Devices, Loading a Device Driver in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Writing an Undefine Method

This article describes how an Undefine device method works. It also suggests guidelines for programmers writing their own Undefine device configuration methods.

Syntax

und*Dev* **-l** *Name*

Description

The Undefine method deletes a Defined device from the Customized database. Once a device is deleted, it cannot be configured until it is once again defined by the Define method.

The Undefine method is also responsible for releasing the major and minor number assignments for the device instance and deleting the device's special files from the **/dev** directory. If minor number assignments are registered with the **genminor** subroutine, the Undefine method can release the major and minor number assignments and delete the special files by using the **reldevno** subroutine.

By convention, the first three characters of the name of the Undefine method are **und**. The remainder of the name (*Dev*) can be any characters, subject to operating system file-name restrictions, that identify the device or group of devices that use the method.

Flags

-l *Name* Identifies the logical name of the device to be undefined.

Guidelines for Writing an Undefine Method

This list of tasks is intended as a guideline for writing an Undefine method. Some devices may have specials needs that are not addressed in these tasks.

The Undefine method must:

1. Validate the input parameters. The **-l** flag must be supplied to identify the device to be undefined.
2. Initialize the Object Data Manager (ODM) using the **odm_initialize** subroutine and lock the configuration database using the **odm_lock** subroutine. See ″Writing a Device Method″ in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts* for an example.
3. Retrieve the Customized Device (CuDv) object for the device to be undefined. This is done by getting the CuDv object whose Device Name descriptor matches the name supplied with the **-l** flag. If no object is found with the specified name, this method exits with an error.
4. Check the device's current state. If the Device Status descriptor indicates that the device is not in the Defined state, then it is not ready to be undefined. If this is the case, this method exits with an error.
5. Check for any child devices. This check is accomplished by querying the CuDv object class for any objects whose Parent Device Logical Name descriptor matches this device's name. If the device has child devices, regardless of the states they are in, the Undefine method will fail. All child devices must be undefined before the parent can be undefined.
6. Check to see if this device is listed as a dependency of another device. This is done by querying the Customized Dependency (CuDep) object class for objects whose Dependency descriptor matches this device's logical name. If a match is found, the method exits with an error. A device may not be undefined if it has been listed as a dependent of another device. ″Understanding Device Dependencies and Child Devices″ in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts* discusses dependencies.

7. Delete Special Files and major and minor numbers. If no errors have been encountered, the method can delete customized information. First, delete the special files from the **/dev** directory. Next, delete all minor number assignments. If the last minor number has been deleted for a particular major number, release the major number as well, using the **relmajor** subroutine. The Undefine method should never delete objects from the Customized Device Driver (CuDvDr) object class directly, but should always use the routines provided. If the minor number assignments are registered with the **genminor** subroutine, all of the above can be accomplished using the **reldevno** subroutine.
8. Delete all attributes for the device from the Customized Attribute (CuAt) object class. Simply delete all CuAt objects whose Device Name descriptor matches this device's logical name. It is not an error if the ODM routines used to delete the attributes indicate that no objects were deleted. This indicates that the device has no attributes that have been changed from the default values.
9. Delete the Customized VPD (CuVPD) object for the device, if it has one.
10. Delete the Customized Dependency (CuDep) objects that indicate other devices that are dependents of this device.
11. Delete the Customized Device (CuDv) object for the device.
12. Close all object classes and terminate the ODM. Exit with an exit code of 0 (zero) if there are no errors.

Files

/dev directory	Contains the device special files.

Related Information

The **genminor** subroutine, **odm_initialize** subroutine, **odm_lock** subroutine, **reldevno** subroutine, **relmajor** subroutine.

Writing a Define Method .

Customized Devices (CuDv) object class, Predefined Devices (PdDv) object class, Customized Attributes (CuAt) object class, Predefined Attribute (PdAt) object class, Customized Vital Product Data (CuVPD) object class.

Understanding Device Dependencies and Child Devices in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Object Data Manager (ODM) Overview for Programmers in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

Special Files Overview in *AIX 5L Version 5.2 Files Reference*.

Device Methods for Adapter Cards: Guidelines

The device methods for an adapter card are essentially the same as for any other device. They need to perform roughly the same tasks as those described in ″Writing a Device Method″ in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*. However, there is one additional important consideration. The Bus Configure method, or Bus Configurator, is responsible for discovering the adapter cards present in the system and for assigning bus resources to each of the adapters. These resources include interrupt levels, DMA arbitration levels, bus memory, and bus I/O space.

Adapters are typically defined and configured at boot time. However, if an adapter is not configured due to unresolvable bus resource conflicts, or if an adapter is unconfigured at run time with the **rmdev** command, an attempt to configure an adapter at run time may occur.

If an attempt is made, the Configure method for the adapter must take these steps to ensure system integrity:

1. Ensure the card is present in the system by reading the POS(0) and POS(1) registers from the slot that is supposed to contain the card and comparing these values with what they are supposed to be for the card.
2. Invoke the **busresolve** subroutine to ensure that the adapter's bus resource attributes, as represented in the database, do not conflict with any of the configured adapters.

Additional guidelines must be followed when adding support for a new adapter card. They are discussed in:

- Adapter-Specific Considerations for the Predefined Attributes (PdAt) object class
- Writing a Configure Method
- Adapter-Specific Considerations for the Predefined Devices (PdDv) object class

Related Information

ODM Device Configuration Object Classes.

The **rmdev** command.

Understanding Direct Memory Access (DMA), Understanding Interrupts in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Machine Device Driver

The machine device driver provides an interface to platform-specific hardware for the system configuration and reliability, availability, and serviceability (RAS) subsystems. The machine device driver supports these special files for accessing this hardware from user mode: **/dev/nvram** and **/dev/bus0** ... **/dev/bus***N* where *N* is the bus number. The **/dev/nvram** special file provides access to special nonvolatile random access memory (RAM) for the purposes of storing or retrieving error information and system boot information. The **/dev/bus***N* special files provide access to the I/O buses for system configuration and diagnostic purposes. The presence and use of this device driver and its associated special files are platform-specific and should not be used by general applications.

A program must have the appropriate privilege to open special files **/dev/nvram** or **/dev/bus***N*. For AIX 4.2.1 and later, it must also have the appropriate privilege to open Common Hardware Reference Platform (CHRP) bus special files **/dev/pci***N*, or **/dev/isa***N*.

Driver Initialization and Termination

Special initialization and termination requirements do not exist for the machine device driver. This driver is statically bound to the operating system kernel and is initialized during kernel initialization. This device driver does not support termination and cannot be unloaded.

/dev/nvram Special File Support

open and close Subroutines

The machine device driver supports the **/dev/nvram** special file as a multiplexed character special file. This special file and the support for NVRAM is provided exclusively on this hardware platform to support the system configuration and RAS subsystems. These subsystems open the **/dev/nvram/***n* special file with a channel name, *n*, specifying the data area to be accessed. An exception is the **/dev/nvram** file with no channel specified, which provides access to general NVRAM control functions and the LED display on the front panel.

A special channel name of **base** can be used to read the base customize information stored as part of the boot record. This information was originally copied to the disk by the **savebase** command and is only copied by the driver at boot time. The **base** customize information can be read only once. When the **/dev/nvram/base** file is closed for the first time, the buffer containing the base customize information is freed. Subsequent opens will return an **ENOENT** error code.

read and write Subroutines

The **write** subroutine is not supported and will return an **ENODEV** error code. The **read** subroutine is supported after a successful open of the **base** channel only. The **read** subroutine transfers data from the data area associated with the specified channel. The transfer starts at the offset (within the channel's data area) specified by the `offset` field associated with the file pointer used on the subroutine call.

On a **read** subroutine, if the end of the data area is reached before the transfer count is reached, the number of bytes read before the end of the data area was reached is returned. If the **read** subroutine starts at the end of the data area, zero bytes are read. If the **read** subroutine starts after the end of the data area, an **ENXIO** error code is returned by the driver.

The **lseek** subroutine can be used to change the starting data-area offset to be used on a subsequent **read** call.

ioctl Operations

The following ioctl operations can be issued to the machine device driver after a successful open of the **/dev/nvram/** special file:

Operation	Description
IOCINFO	Returns machine device driver information in the caller's **devinfo** structure (pointed to by the *arg* parameter). This structure is defined in the **/usr/include/sys/devinfo.h** file. The device type for this device driver is **DD_PSEU**.
MIOGETKEY	Returns the status of the keylock. The *arg* parameter should point to a **mach_dd_io** structure. The `md_data` field should point to an integer; this contains the status of the keylock on return. **Note:** Not all platforms have a physical keylock that software can read. For these platforms, status is established at boot time.
MIOGETPS	Returns the power status. The *arg* parameter should point to a **mach_dd_io** structure. The `md_data` field should point to an integer; this contains the power status on return. **Note:** Not all platforms provide power status.
MIOIPLCB	Returns the contents of the boot control block. The *arg* parameter is set to point to a **mach_dd_io** structure, which describes the data area where the boot control block is to be placed. The format of this control block is specified in the **/usr/include/sys/iplcb.h** file and the **mach_dd_io** structure is defined in the **/usr/include/sys/mdio.h** file. This **ioctl** operation uses the following fields in the **mach_dd_io** structure: **md_data** Points to a buffer at least the size of the value in the `md_size` field. **md_size** Specifies the size (in bytes) of the buffer pointed to by the `md_data` field and is the number of bytes to be returned from the boot control block. **md_addr** Specifies an offset into the boot control block where data is to be obtained. **Note:** Regions within this control block are platform dependent.

Operation	Description
MIONVGET	Reads data from an NVRAM address and returns data in the buffer provided by the caller. This is useful for reading the ROS area of NVRAM. A structure defining this area is in the **/usr/include/sys/mdio.h** file. Use of this ioctl operation is not supported for systems that are compliant with the PowerPC Reference Platform or the Common Hardware Reference Platform and, in AIX 4.2.1 and later, cause the operation to fail with an **EINVAL** error code.
MIONVLED	Writes the value found in the *arg* parameter to the system front panel LED display. On this panel, three digits are available and the *arg* parameter value can range from 0 to hex FFF. An explanation of the LED codes can be found in the **/usr/include/sys/mdio.h** file. **Note:** Not all platforms have an LED.
MIONVPUT	Writes data to an NVRAM address from the buffer provided by the caller. This operation is used only to update the ROS area of NVRAM and only by system commands. Use of this operation in other areas of NVRAM can cause unpredictable results to occur. If the NVRAM address provided is within the ROS area, a new cyclic redundancy code (CRC) for the ROS area is generated. Use of this ioctl operation is not supported on systems that are compliant with the PowerPC Reference Platform or the Common Hardware Reference Platform and, in AIX 4.2.1 and later, cause the operation to fail with an **EINVAL** error code.

ioctl Operations for POWER-based Systems

The following four ioctl operations can be used only with the POWER-based architecture. If used with other systems, or if an illegal offset address, size, or slot number is supplied, these operations return an **EINVAL** error code.

These ioctls can be called from user space or kernel space (using the **fp_ioctl** kernel service), but they are available only in the process environment.

The ioctl argument must be a pointer to a **mach_dd_io** structure.

The lock will be obtained to serialize access to the bus slot configuration register.

MIOVPDGET: This ioctl allows read access to VPD/ROM address space.

The following structure members must be supplied:

Structure Member	Description
ulong `md_addr`	Specifies the offset into the feature/VPD address space to begin reading.
ulong `md_size`	Specifies the number of bytes to be transferred.
char `md_data`	Specifies a pointer to user buffer for data.
int `md_sla`	Specifies a slot number (bus slot configuration select).
int `md_incr`	Requires byte access (**MV_BYTE**).

The read begins at base address 0xFFA00000. The offset will be added to the base address to obtain the starting address for reading.

The **buc_info** structure for the selected bus slot will be used to obtain the word increment value. This value performs correct addressing while reading the data.

MIOCFGGET: This ioctl allows read access to the architected configuration registers.

The following structure members must be supplied:

Structure Member	Description
ulong md_addr	Specifies the offset into the configuration register address space.
ulong md_size	Specifies a value of 1.
char md_data	Specifies a pointer to user buffer for data.
int md_sla	Specifies a slot number (bus slot configuration select).
int md_incr	Requires byte or word access (**MV_BYTE**/**MV_SHORT**/**MV_WORD**).

The base address 0xFF200000 will be added to the offset to obtain the address for the read.

MIOCFGPUT: This ioctl allows write access to the architected configuration registers.

The following structure members must be supplied:

Structure Member	Description
ulong md_addr	Specifies the offset into the configuration register address space.
ulong md_size	Specifies a value of 1.
char md_data	Specifies a pointer to user buffer of data to write.
int md_sla	Specifies a slot number (bus slot configuration select).
int md_incr	Requires byte or word access (**MV_BYTE**/**MV_SHORT**/**MV_WORD**).

The base address 0xFF200000 will be added to the offset to obtain the address for the read.

MIORESET: This ioctl allows access to the architected bus slot reset register.

The following structure members must be supplied:

Structure Member	Description
ulong md_addr	Specifies reset hold time (in nanoseconds).
ulong md_size	Not used.
char md_data	Not used.
int md_sla	Specifies a slot number (bus slot configuration select).
int md_incr	Not used.

The bus slot reset register bit corresponding to the specified bus slot is set to 0. After the specified delay, the bit is set back to 1 and control returns to the calling program.

If a reset hold time of 0 is passed, the bus slot remains reset on return to the calling process.

ioctl Operations for the PowerPC Reference Platform Specification and the Common Hardware Reference Platform

The following four ioctl operations can be used only with the PowerPC Reference Platform and, in AIX 4.2.1 and later, the Common Hardware Reference Platform.

MIOGEARD: Scans for the variable name in the Global Environment Area, and, if found, the null terminated string will be returned to the caller. A global variable is of the form "variablename=". The returned string is of the form "variablename=string". If the supplied global variable is "*=", all of the variable strings in the Global Environment Area will be returned.

The following structure members must be supplied:

Structure Member	Description
ulong md_addr	Pointer to global variable string which is null terminated with an equal sign as the last non-null character.

Structure Member	Description
ulong md_size	Number of bytes in data buffer.
int md_incr	Not used.
char md_data	Pointer to the data buffer.
int md_sla	Not used.
ulong md_length	This is a pointer to the length of the returned global variable string(s) including the null terminator(s) if md_length is non-zero.

MIOGEAUPD: The specified global variable will be added to the Global Environment Area if it does not exist. If the specified variable does exist in the Global Environment Area, the new contents will replace the old after making adjustments for any size deltas. Further, any information moved toward a lower address will have the original area zeroed. If there is no string following the variable name and equal sign, the specified variable will be deleted. If the variable to be deleted is not found, a successful return will occur. The new information will be written to **NVRAM**. Further, the header in **NVRAM** will be updated to include the update time of the Global Environment Area and the Crc1 value will be recomputed.

The following structure members must be supplied:

Structure Member	Description
ulong md_addr	Pointer to global variable string which is null terminated.
ulong md_size	Not used.
int md_incr	Not used.
char md_data	Not used.
int md_sla	Not used.
ulong md_length	This is a pointer to the amount of space left in the Global Environment Area after the update. This is computed as the size of the area minus the length of all global variable strings minus the threshold value.

MIOGEAST: The specified threshold will be set so that any updates done will not exceed the Global Environment Area size minus the threshold. In place of the the **mdio** structure an integer value is used to specify the threshold. The threshold does not persist across IPLs.

MIOGEARDA: The attributes of the Global Environment Area will be returned to the data area specified by the caller. The **gea_attrib** structure has been added to **mdio.h**. It contains the following information:

Structure Member	Description
long gea_length	number of bytes in the Global Environment Area of **NVRAM**.
long gea_used	number of bytes used in the Global Environment Area.
long gea_thresh	Global Environment Area threshold value.
ulong md_addr	Not used.
ulong md_size	Size of the data buffer. It must be greater than or equal to the size of (**gea_attrib**).
int md_incr	Not used.
char md_data	Address of the buffer to copy the **gea_attrib** structure.
int md_sla	Not used.
ulong md_length	Not used.

MIONVPARTLEN: The length of the CHRP **NVRAM** partition will be returned to the data area specified by the caller. The following structure members must be supplied:

Structure Member	Description
ulong md_addr	Specifies the partition signature.
ulong *md_length	Specifies a pointer to the name of the partition.
int md_incr	Not used.

Structure Member	Description
ulong md_size	Specifies the data area for the returned partition length.
char *md_data	Not used.
int md_sla	Not used.

MIONVPARTRD: **MIONVPARTRD** performs read actions on CHRP **NVRAM** partitions. The following structure members must be supplied:

Structure Member	Description
ulong md_addr	Specifies the partition signature.
ulong *md_length	Specifies a pointer to the name of the partition.
int md_incr	Specifies the start offset into the partition.
ulong md_size	Specifies the number of bytes to be read.
char *md_data	Specifies a pointer to the user buffer where data will be copied.
int md_sla	Not used.

MIONVPARTUPD: **MIONVPARTUPD** performs write actions to CHRP **NVRAM** partitions. The following structure members must be supplied:

Structure Member	Description
ulong md_addr	Specifies the partition signature.
ulong *md_length	Specifies a pointer to the name of the partition.
int md_incr	Specifies the start offset into the partition.
ulong md_size	Specifies the number of bytes to be read.
char *md_data	Specifies a pointer to the user buffer for data to write.
int md_sla	Not used.

Error Codes

The following error conditions may be returned when accessing the machine device driver with the **/dev/nvram/***n* special file:

Error Condition	Description
EACCES	A write was requested to a file opened for read access only.
ENOENT	An open of **/dev/nvram/base** was attempted after the first close.
EFAULT	A buffer specified by the caller was invalid on a **read**, **write**, or **ioctl** subroutine call.
EINVAL	An invalid **ioctl** operation was issued.
ENXIO	A read was attempted past the end of the data area specified by the channel.
ENODEV	A write was attempted.
ENOMEM	A request was made with a user-supplied buffer that is too small for the requested data or not enough memory could be allocated to complete the request.

Bus Special File Support

All models have at least one bus. For non-CHRP systems, the names are of the form **/dev/bus***N*. CHRP systems will have the form **/dev/pci***N* and **/dev/isa***N*.

open and close Subroutines

The machine device driver supports the bus special files as character special files. These special files, and support for access to the I/O buses and controllers, are provided on this hardware platform to support the system configuration and diagnostic subsystems, exclusively. The configuration subsystem accesses the I/O buses and controllers through the machine device driver to determine the I/O configuration of the system. This driver can also be used to configure the I/O controllers and devices as required for proper

system operation. If the system diagnostics are unable to access a device through the diagnostic functions provided by the device's own device driver, they may use the machine device driver to attempt further failure isolation.

read and write Subroutines

The **read** and **write** subroutines are not supported by the machine device driver through the bus special files and, if called, return an **ENOENT** return code in the **errno** global variable.

ioctl Operations

The bus ioctl operations allow transfers of data between the system I/O controller or the system I/O bus and a caller-supplied data area. Because these **ioctl** operations use the **mach_dd_io** structure, the *arg* parameter on the **ioctl** subroutine must point to such a structure. The bus address, the pointer to the caller's buffer, and the number and length of the transfer are all specified in the **mach_dd_io** structure. The **mach_dd_io** structure is defined in the **/usr/include/sys/mdio.h** file and provides the following information:

- The `md_addr` field contains an I/O controller or I/O bus address.
- The `md_data` field points to a buffer at least the size of the value in the `md_size` field.
- The `md_size` field contains the number of items to be transferred.
- The `md_incr` field specifies the length of the transferred item. It must be set to **MV_BYTE**, **MV_SHORT**, or **MV_WORD.**

The following commands can be issued to the machine device driver after a successful open of the bus special file:

Command	Description
IOCINFO	Returns machine device driver information in the caller's **devinfo** structure, as specified by the *arg* parameter. This structure is defined in the **/usr/include/sys/devinfo.h** file. The device type for this device driver is **DD_PSEU**.
MIOBUSGET	Reads data from the bus I/O space and returns it in a caller-provided buffer.
MIOBUSPUT	Writes data supplied in the caller's buffer to the bus I/O space.
MIOMEMGET	Reads data from the bus memory space and returns it to the caller-provided buffer.
MIOMEMPUT	Writes data supplied in the caller-provided buffer to the bus memory space.
MIOPCFGET	Reads data from the PCI bus configuration space and returns it in a caller-provided buffer. The **mach_dd_io** structure field **md_sla** must contain the Device Number and Function Number for the device to be accessed.
MIOPCFPUT	Writes data supplied in the caller's buffer to the PCI bus configuration space. The **mach_dd_io** structure field **md_sla** must contain the Device Number and Function Number for the device to be accessed.

Error Codes

EFAULT	A buffer specified by the caller was invalid on an **ioctl** call.
EIO	An unrecoverable I/O error occurred on the requested data transfer.
ENOMEM	No memory could be allocated by the machine device driver for use in the data transfer.

Files

/dev/pci*N*	Provides access to the I/O bus (CHRP only, AIX 4.2.1 and later).
/dev/isa*N*	Provides access to the I/O bus (CHRP only, AIX 4.2.1 and later).
/dev/nvram	Provides access to platform-specific nonvolatile RAM.
/dev/nvram/base	Allows read access to the base customize information stored as part of the boot record.

Related Information

The **close** subroutine, **ioctl** subroutine, **lseek** subroutine, **open** subroutine, **read** subroutine, **write** subroutine.

The **savebase** device configuration command.

The **bus** special file, **nvram** special file.

Loading a Device Driver

The **loadext** subroutine is used to load and unload device drivers. The name of the device driver is passed as a parameter to the **loadext** routine. If the device driver is located in the **/usr/lib/drivers** directory, just the device driver name without path information can be specified to the **loadext** subroutine. If the device driver is located in another directory, the fully qualified path name of the device driver must be specified.

The Device Driver Name descriptor of Predefined Devices (PdDv) object class objects is intended to contain only the device driver name and not the fully qualified path name. For device drivers located in the **/usr/lib/drivers** directory, a Configure method can obtain the name of the driver from the Device Driver Name descriptor to pass to the **loadext** routine. This is convenient since most drivers are located in the **/usr/lib/drivers** directory.

If a device driver is located in a directory other than the **/usr/lib/drivers** directory, the path name must be handled differently. The Configure method could be coded to assume a particular path name, or for more flexibility the path name could be stored as an attribute in the Predefined Attribute (PdAt) object class. The Configure method is responsible for knowing how to obtain the fully qualified path name to pass to the **loadext** subroutine.

Files

/usr/lib/drivers directory	Contains device drivers.

Related Information

The **loadext** subroutine.

Predefined Devices (PdDv) object class, Predefined Attribute (PdAt) object class.

Writing a Configure Method .

How Device Methods Return Errors

Device methods indicate errors to the Configuration Manager and run-time configuration commands by exiting with a nonzero exit code. The Configuration Manager and configuration commands can understand only the exit codes defined in the **cf.h** file.

More than one error code can describe a given error. This is because many exit codes correspond to highly specific errors, while others are more general. Whenever possible, use the most specific error code possible.

For example, if your Configure method obtains an attribute from the Customized Attributes (CuAt) object class for filling in the device-dependent structure (DDS), but the value is invalid (possibly due to a corrupted database), you might exit with an **E_BADATTR** error. Otherwise, you might choose the **E_DDS** exit code, due to another error condition that occurred while building the DDS.

Related Information

ODM Device Configuration Object Classes.

The Customized Attributes (CuAt) object class.

The Device Dependent Structure (DDS) Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

ODM Device Configuration Object Classes

A list of the ODM Device Configuration Object Classes follows:

PdDv	Predefined Devices
PdCn	Predefined Connection
PdAt	Predefined Attribute
Config_Rules	Configuration Rules
CuDv	Customized Devices
CuDep	Customized Dependency
CuAt	Customized Attribute
CuDvDr	Customized Device Driver
CuVPD	Customized Vital Product Data

Related Information

Device Configuration Subsystem Programming Introduction, Writing a Device Method in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Configuration Rules (Config_Rules) Object Class

Description

The Configuration Rules (Config_Rules) object class contains the configuration rules used by the Configuration Manager. The Configuration Manager runs in two phases during system boot. The first phase is responsible for configuring the base devices so that the real root device can be configured and made ready for operation. The second phase configures the rest of the devices in the system after the root file system is up and running. The Configuration Manager can also be invoked at run time. The Configuration Manager routine is driven by the rules in the Config_Rules object class.

The Config_Rules object class is preloaded with predefined configuration rules when the system is delivered. There are three types of rules: phase 1, phase 2, and phase 2 service. You can use the ODM commands to add, remove, change, and show new or existing configuration rules in this object class to customize the behavior of the Configuration Manager. However, any changes to a phase 1 rule must be written to the boot file system to be effective. This is done with the **bosboot** command.

All logical and physical devices in the system are organized in clusters of tree structures called nodes. For information on nodes or tree structures, see the ″Device Configuration Manager Overview″ in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*. The rules in the Config_Rules object class specify program names that the Configuration Manager executes. Usually, these programs are the configuration programs for the top of the nodes. When these programs are invoked, the names of the next lower-level devices that need to be configured are returned in standard output.

The Configuration Manager configures the next lower-level devices by invoking the Configure method for those devices. In turn, those devices return a list of device names to be configured. This process is repeated until no more device names are returned. All devices in the same node are configured in a transverse order.

The second phase of system boot requires two sets of rules: phase 2 and service. The position of the key on the front panel determines which set of rules is used. The service rules are used when the key is in the service position. If the key is in any other position, the phase 2 rules are used. Different types of rules are indicated in the Configuration Manager Phase descriptor of this object class.

Each configuration rule has an associated boot mask. If this mask has a nonzero value, it represents the type of boot to which the rule applies. For example, if the mask has a **DISK_BOOT** value, the rule applies to system boots where disks are base devices. The type of boot masks are defined in the **/usr/include/sys/cfgdb.h** file.

Descriptors

The **Config_Rules** object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_SHORT	**phase**	Configuration Manager Phase	Required
ODM_SHORT	**seq**	Sequence Value	Required
ODM_LONG	**boot_mask**	Type of boot	Required
ODM_VCHAR	**rule_value[RULESIZE]**	Rule Value	Required

These descriptors are described as follows:

Descriptor — **Description**

Configuration Manager Phase

This descriptor indicates which phase a rule should be executed under phase 1, phase 2, or phase 2 service.

- **1** Indicates that the rule should be executed in phase 1.
- **2** Indicates that the rule should be executed in phase 2.
- **3** Indicates that the rule should be executed in phase 2 service mode.

Sequence Value

In relation to the other rules of this phase, the `seq` number indicates the order in which to execute this program. In general, the lower the `seq` number, the higher the priority. For example, a rule with a `seq` number of 2 is executed before a rule with a `seq` number of 5. There is one exception to this: a value of 0 indicates a DONT_CARE condition, and any rule with a `seq` number of 0 is executed last.

Type of boot

If the `boot_mask` field has a nonzero value, it represents the type of boot to which the rule applies. If the **-m** flag is used when invoking the **cfgmgr** command, the **cfgmgr** command applies the specified mask to this field to determine whether to execute the rule. By default, the **cfgmgr** command always executes a rule for which the `boot_mask` field has a 0 value.

Descriptor	Description
Rule Value	This is the full path name of the program to be invoked. The rule value descriptor may also contain any options that should be passed to that program. However, options must follow the program name, as the whole string will be executed as if it has been typed in on the command line. **Note:** There is one rule for each program to execute. If multiple programs are needed, then multiple rules must be added.

Rule Values

Phase	Sequence	Type of boot	Rule Value
1	1	0	/usr/lib/methods/defsys
1	10	0x0001	/usr/lib/methods/deflvm
2	1	0	/usr/lib/methods/defsys
2	5	0	/usr/lib/methods/ptynode
2	10	0	/usr/lib/methods/starthft
2	15	0	/usr/lib/methods/starttty
2	20	0x0010	/usr/lib/methods/rc.net
3	1	0	/usr/lib/methods/defsys
3	5	0	/usr/lib/methods/ptynode
3	10	0	/usr/lib/methods/starthft
3	15	0	/usr/lib/methods/starttty

Related Information

The **bosboot** command.

Writing a Configure Method .

Writing a Device Method, Device Configuration Manager Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Device Configuration Subsystem Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Object Data Management (ODM) Overview for Programmers, Understanding ODM Object Classes and Objects in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

Understanding System Boot Processing in *AIX 5L Version 5.2 System Management Concepts: Operating System and Devices*.

Customized Attribute (CuAt) Object Class

Description

The Customized Attribute (CuAt) object class contains customized device-specific attribute information.

Device instances represented in the Customized Devices (CuDv) object class have attributes found in either the Predefined Attribute (PdAt) object class or the CuAt object class. There is an entry in the CuAt object class for attributes that take nondefault values. Attributes taking the default value are found in the PdAt object class. Each entry describes the current value of the attribute.

When changing the value of an attribute, the Predefined Attribute object class must be referenced to determine other possible attribute values.

Both attribute object classes must be queried to get a complete set of current values for a particular device's attributes. Use the **getattr** and **putattr** subroutines to retrieve and modify, respectively, customized attributes.

Descriptors

The Customized Attribute object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	**name[NAMESIZE]**	Device Name	Required
ODM_CHAR	**attribute[ATTRNAMESIZE]**	Attribute Name	Required
ODM_VCHAR	**value[ATTRVALSIZE]**	Attribute Value	Required
ODM_CHAR	**type[FLAGSIZE]**	Attribute Type	Required
ODM_CHAR	**generic[FLAGSIZE]**	Generic Attribute Flags	Optional
ODM_CHAR	**rep[FLAGSIZE]**	Attribute Representation Flags	Required
ODM_SHORT	**nls_index**	NLS Index	Optional

These descriptors are described as follows:

Descriptor	Description
Device Name	Identifies the logical name of the device instance to which this attribute is associated.
Attribute Name	Identifies the name of a customized device attribute.
Attribute Value	Identifies a customized value associated with the corresponding Attribute Name. This value is a nondefault value.
Attribute Type	Identifies the attribute type associated with the Attribute Name. This descriptor is copied from the Attribute Type descriptor in the corresponding PdAt object when the CuAt object is created.
Generic Attribute Flags	Identifies the Generic Attribute flag or flags associated with the Attribute Name. This descriptor is copied from the Generic Attribute Flags descriptor in the corresponding PdAt object when the CuAt object is created.
Attribute Representation Flags	Identifies the Attribute Value's representation. This descriptor is copied from the Attribute Representation flags descriptor in the corresponding Predefined Attribute object when the Customized Attribute object is created.
NLS Index	Identifies the message number in the NLS message catalog that contains a textual description of the attribute. This descriptor is copied from the NLS Index descriptor in the corresponding Predefined Attribute object when the Customized Attribute object is created.

Related Information

ODM Device Configuration Object Classes.

Customized Devices (CuDv) object class, Predefined Attribute (PdAt) object class.

The **getattr** device configuration subroutine, **putattr** device configuration subroutine.

List of Device Configuration Subroutines in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Customized Dependency (CuDep) Object Class

Description

The Customized Dependency (CuDep) object class describes device instances that depend on other device instances. Dependency does not imply a physical connection. This object class describes the dependence links between logical devices and physical devices as well as dependence links between logical devices, exclusively. Physical dependencies of one device on another device are recorded in the Customized Device (CuDev) object class.

Descriptors

The Customized Dependency object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	**name[NAMESIZE]**	Device Name	Required
ODM_CHAR	**dependency[NAMESIZE]**	Dependency (device logical name)	Required

These descriptors are described as follows:

Descriptor **Description**

Device Name Identifies the logical name of the device having a dependency.

Dependency Identifies the logical name of the device instance on which there is a dependency. For example, a mouse, keyboard, and display might all be dependencies of a device instance of `lft0`.

Related Information

ODM Device Configuration Object Classes.

Customized Device (CuDv) object class.

Customized Device Driver (CuDvDr) Object Class

Description

The Customized Device Driver (CuDvDr) object class stores information about critical resources that need concurrence management through the use of the Device Configuration Library subroutines. You should only access this object class through these five Device Configuration Library subroutines: the **genmajor**, **genminor**, **relmajor**, **reldevno**, and **getminor** subroutines.

These subroutines exclusively lock this class so that accesses to it are serialized. The **genmajor** and **genminor** routines return the major and minor number, respectively, to the calling method. Similarly, the **reldevno** and **relmajor** routines release the major or minor number, respectively, from this object class.

Descriptors

The Customized Device Driver object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	**resource[RESOURCESIZE]**	Resource Name	Required

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	**value1[VALUESIZE]**	Value1	Required
ODM_CHAR	**value2[VALUESIZE]**	Value2	Required
ODM_CHAR	**value3[VALUESIZE]**	Value3	Required

The Resource descriptor determines the nature of the values in the Value1, Value2, and Value3 descriptors. Possible values for the Resource Name descriptor are the strings **devno** and **ddins**.

The following table specifies the contents of the Value1, Value2, and Value3 descriptors, depending on the contents of the Resource Name descriptor.

Resource	Value1	Value2	Value3
devno	Major number	Minor number	Device instance name
ddins	Dd instance name	Major number	Null string

When the Resource Name descriptor contains the **devno** string, the `Value1` field contains the device major number, `Value2` the device minor number, and `Value3` the device instance name. These value descriptors are filled in by the **genminor** subroutine, which takes a major number and device instance name as input and generates the minor number and resulting **devno** Customized Device Driver object.

When the Resource Name descriptor contains the **ddins** string, the `Value1` field contains the device driver instance name. This is typically the device driver name obtained from the Device Driver Name descriptor of the Predefined Device object. However, this name can be any unique string and is used by device methods to obtain the device driver major number. The `Value2` field contains the device major number and the `Value3` field is not used. These value descriptors are set by the **genmajor** subroutine, which takes a device instance name as input and generates the corresponding major number and resulting **ddins** Customized Device Driver object.

Related Information

ODM Device Configuration Object Classes.

Predefined Devices (PdDv) object class.

The **genmajor** device configuration subroutine, **genminor** device configuration subroutine, **getminor** device configuration subroutine, **reldevno** device configuration subroutine, **relmajor** device configuration subroutine.

List of Device Configuration Subroutines in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Customized Devices (CuDv) Object Class

Description

The Customized Devices (CuDv) object class contains entries for all device instances defined in the system. As the name implies, a defined device object is an object that a Define method has created in the CuDv object class. A defined device instance may or may not have a corresponding actual device attached to the system.

A CuDv object contains attributes and connections specific to the device instance. Each device instance, distinguished by a unique logical name, is represented by an object in the CuDv object class. The

Customized database is updated twice, during system boot and at run time, to define new devices, remove undefined devices, or update the information for a device whose attributes have been changed.

Descriptors

The Customized Devices object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	**name[NAMESIZE]**	Device Name	Required
ODM_SHORT	**status**	Device Status Flag	Required
ODM_SHORT	**chgstatus**	Change Status Flag	Required
ODM_CHAR	**ddins[TYPESIZE]**	Device Driver Instance	Optional
ODM_CHAR	**location[LOCSIZE]**	Location Code	Optional
ODM_CHAR	**parent[NAMESIZE]**	Parent Device Logical Name	Optional
ODM_CHAR	**connwhere[LOCSIZE]**	Location Where Device Is Connected	Optional
ODM_LINK	**PdDvLn**	Link to Predefined Devices Object Class	Required

These descriptors are described as follows:

Descriptor **Description**

Device Name

A Customized Device object for a device instance is assigned a unique logical name to distinguish the instance from other device instances. The device logical name of a device instance is derived during Define method processing. The rules for deriving a device logical name are:

- The name should start with a *prefix name* pre-assigned to the device instance's associated device type. The prefix name can be retrieved from the Prefix Name descriptor in the Predefined Device object associated with the device type.
- To complete the logical device name, a *sequence number* is usually appended to the prefix name. This sequence number is unique among all defined device instances using the same prefix name. Use the following subrules when generating sequence numbers:
  - A sequence number is a non-negative integer represented in character format. Therefore, the smallest available sequence number is 0.
  - The next available sequence number relative to a given prefix name should be allocated when deriving a device instance logical name.
  - The next available sequence number relative to a given prefix name is defined to be the smallest sequence number not yet allocated to defined device instances using the same prefix name.

    For example, if `tty0`, `tty1`, `tty3`, `tty5`, and `tty6` are currently assigned to defined device instances, then the next available sequence number for a device instance with the `tty` prefix name is `2`. This results in a logical device name of `tty2`.

The **genseq** subroutine can be used by a Define method to obtain the next available sequence number.

Descriptor	Description
Device Status Flag	Identifies the current status of the device instance. The device methods are responsible for setting Device Status flags for device instances. When the Define method defines a device instance, the device's status is set to `defined`. When the Configure method configures a device instance, the device's status is typically set to `available`. The Configure method takes a device to the Stopped state only if the device supports the Stopped state. When the Start method starts a device instance, its device status is changed from the Stopped state to the Available state. Applying a Stop method on a started device instance changes the device status from the Available state to the Stopped state. Applying an Unconfigure method on a configured device instance changes the device status from the Available state to the Defined state. If the device supports the Stopped state, the Unconfigure method takes the device from the Stopped state to the Defined state. "Understanding Device States" in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts* provides more information about the Available, Defined, and Stopped states. The possible status values are: **DEFINED** Identifies a device instance in the Defined state. **AVAILABLE** Identifies a device instance in the Available state. **STOPPED** Identifies a device instance in the Stopped state.
Change Status Flag	This flag tells whether the device instance has been altered since the last system boot. The diagnostics facility uses this flag to validate system configuration. The flag can take these values: **NEW** Specifies whether the device instance is new to the current system boot. **DONT_CARE** Identifies the device as one whose presence or uniqueness cannot be determined. For these devices, the new, same, and missing states have no meaning. **SAME** Specifies whether the device instance was known to the system prior to the current system boot. **MISSING** Specifies whether the device instance is missing. This is true if the device is in the CuDv object class, but is not physically present.
Device Driver Instance	This descriptor typically contains the same value as the Device Driver Name descriptor in the Predefined Devices (PdDv) object class if the device driver supports only one major number. For a driver that uses multiple major numbers (for example, the logical volume device driver), unique instance names must be generated for each major number. Since the logical volume uses a different major number for each volume group, the volume group logical names would serve this purpose. This field is filled in with a null string if the device instance does not have a corresponding device driver.

Descriptor	Description
Location Code	Identifies the location code of the device. This field provides a means of identifying physical devices. The location code format is defined as **AB-CD-EF-GH**, where: **AB** Identifies the CPU and Async drawers with a drawer ID. **CD** Identifies the location of an adapter, memory card, or Serial Link Adapter (SLA) with a slot ID. **EF** Identifies the adapter connector that something is attached to with a connector ID. **GH** Identifies a port, device, or field replaceable unit (FRU), with a port or device or FRU ID, respectively. For more information on the location code format, see ″Location Codes″ and ″Devices Overview for System Management″ in *AIX 5L Version 5.2 System Management Concepts: Operating System and Devices*.
Parent Device Logical Name	Identifies the logical name of the parent device instance. In the case of a real device, this indicates the logical name of the parent device to which this device is connected. More generally, the specified parent device is the device whose Configure method is responsible for returning the logical name of this device to the Configuration Manager for configuring this device. This field is filled in with a null string for a node device.
Location Where Device Is Connected	Identifies the specific location on the parent device instance where this device is connected. The term *location* is used in a generic sense. For some device instances such as the operating system bus, location indicates a slot on the bus. For device instances such as the SCSI adapter, the term indicates a logical port (that is, a SCSI ID and Logical Unit Number combination). For example, for a bus device the location can refer to a specific slot on the bus, with values 1, 2, 3 ... . For a multiport serial adapter device, the location can refer to a specific port on the adapter, with values 0, 1, ...
Link to Predefined Devices Object Class	Provides a link to the device instance's predefined information through the Unique Type descriptor in the PdDv object class.

Related Information

ODM Device Configuration Object Classes.

Predefined Devices (PdDv) object class.

The **genseq** subroutine.

Writing a Define Method, Writing a Configure Method , Writing a Change Method , Writing an Undefine Method , Writing an Unconfigure Method , Writing Optional Start and Stop Methods .

The SCSI Adapter Device Driver in *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1*.

Understanding Physical Volumes and the Logical Volume Device Driver in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding Device States, Device Configuration Manager Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Location Codes, Devices Overview for System Management in *AIX 5L Version 5.2 System Management Concepts: Operating System and Devices*.

Customized VPD (CuVPD) Object Class

Description

The Customized Vital Product Data (CuVPD) object class contains the Vital Product Data (VPD) for customized devices. VPD can be either machine-readable VPD or manually entered user VPD information.

Descriptors

The Customized VPD object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	**name[NAMESIZE]**	Device Name	Required
ODM_SHORT	**vpd_type**	VPD Type	Required
ODM_LONGCHAR	**vpd[VPDSIZE]**	VPD	Required

These fields are described as follows:

Descriptor **Description**

Device Name Identifies the device logical name to which this VPD information belongs.

VPD Type Identifies the VPD as either machine-readable or manually-entered. The possible values:

HW_VPD
Identifies machine-readable VPD.

USER_VPD
Identifies manually entered VPD.

VPD Identifies the VPD for the device. For machine-readable VPD, an entry in this field might include such information as serial numbers, engineering change levels, and part numbers.

Related Information

ODM Device Configuration Object Class.

The *Hardware Technical Reference* provides more details on the VPD.

Predefined Attribute (PdAt) Object Class

Description

The Predefined Attribute (PdAt) object class contains an entry for each existing attribute for each device represented in the Predefined Devices (PdDv) object class. An attribute, in this sense, is any device-dependent information not represented in the PdDv object class. This includes information such as interrupt levels, bus I/O address ranges, baud rates, parity settings, block sizes, and microcode file names.

Each object in this object class represents a particular attribute belonging to a particular device class-subclass-type. Each object contains the attribute name, default value, list or range of all possible values, width, flags, and an NLS description. The flags provide further information to describe an attribute.

Note: For a device being defined or configured, only the attributes that take a nondefault value are copied into the Customized Attribute (CuAt) object class. In other words, for a device being customized, if its attribute value is the default value in the PdDv object class, then there will not be an entry for the attribute in the CuAt object class.

Types of Attributes

There are three types of attributes. Most are *regular* attributes, which typically describe a specific attribute of a device. The *group* attribute type provides a grouping of regular attributes. The *shared* attribute type identifies devices that must all share a given attribute.

A shared attribute identifies another regular attribute as one that must be shared. This attribute is always a bus resource. Other regular attributes (for example, bus interrupt levels) can be shared by devices but are not themselves *shared* attributes. *Shared* attributes require that relevant devices have the same values for this attribute. The Attribute Value descriptor for the shared attribute gives the name of the regular attribute that must be shared.

A group attribute specifies a set of other attributes whose values are chosen as the group, as well as the group attribute number used to choose default values. Each attribute listed within a group has an associated list of possible values it can take. These values must be represented as a list, not as a range. For each attribute within the group, the list of possible values must also have the same number of choices. For example, if the possible number of values is *n*, the group attribute number itself can range from 0 to *n*-1. The particular value chosen for the group indicates the value to pick for each of the attributes in the group. For example, if the group attribute number is 0, then the value for each of the attributes in the group is the first value from their respective lists.

Predefined Attribute Object Class Descriptors

The Predefined Attribute object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	**uniquetype[UNIQUESIZE]**	Unique Type	Required
ODM_CHAR	**attribute[ATTRNAMESIZE]**	Attribute Name	Required
ODM_VCHAR	**deflt[DEFAULTSIZE]**	Default Value	Required
ODM_VCHAR	**values[ATTRVALSIZE]**	Attribute Values	Required
ODM_CHAR	**width[WIDTHSIZE]**	Width	Optional
ODM_CHAR	**type[FLAGSIZE]**	Attribute Type Flags	Required
ODM_CHAR	**generic[FLAGSIZE]**	Generic Attribute Flags	Optional
ODM_CHAR	**rep[FLAGSIZE]**	Attribute Representation Flags	Required
ODM_SHORT	**nls_index**	NLS index	Optional

These descriptors are described as follows:

Descriptor	Description
Unique Type	Identifies the class-subclass-type name of the device to which this attribute is associated. This descriptor is the same as the Unique Type descriptor in the PdDv object class.
Attribute Name	Identifies the name of the device attribute. This is the name that can be passed to the **mkdev** and **chdev** configuration commands and device methods in the attribute-name and attribute-value pairs.

Descriptor	Description
Default Value	If there are several choices or even if there is only one choice for the attribute value, the default is the value to which the attribute is normally set. For groups, the default value is the group attribute number. For example, if the possible number of choices in a group is *n*, the group attribute number is a number between 0 and *n*-1. For shared attributes, the default value is set to a null string. When a device is defined in the system, attributes that take nondefault values are found in the CuAt object class. Attributes that take the default value are found in this object class; these attributes are not copied over to the CuAt object class. Therefore, both attribute object classes must be queried to get a complete set of customized attributes for a particular device.
Attribute Values	Identifies the possible values that can be associated with the attribute name. The format of the value is determined by the attribute representation flags. For regular attributes, the possible values can be represented as a string, hexadecimal, octal, or decimal. In addition, they can be represented as either a range or an enumerated list. If there is only one possible value, then the value can be represented either as a single value or as an enumerated list with one entry. The latter is recommended, since the use of enumerated lists allows the **attrval** subroutine to check that a given value is in fact a possible choice. If the value is hexadecimal, it is prefixed with the 0x notation. If the value is octal, the value is prefixed with a leading zero. If the value is decimal, its value is represented by its significant digits. If the value is a string, the string itself should not have embedded commas, since commas are used to separate items in an enumerated list. A range is represented as a triplet of values: *lowerlimit*, *upperlimit*, and *increment value*. The *lowerlimit* variable indicates the value of the first possible choice. The *upperlimit* variable indicates the value of the last possible choice. The *lowerlimit* and *upperlimit* values are separated by a - (hyphen). Values between the *lowerlimit* and *upperlimit* values are obtained by adding multiples of the *increment value* variable to the *lowerlimit* variable. The *upperlimit* and *increment value* variables are separated by a comma. Only numeric values are used for ranges. Also, discontinuous ranges (for example, 1-3, 6-8) are disallowed. A combination of list and ranges is not allowed. An enumerated list contains values that are comma-separated. If the attribute is a group, the Possible Values descriptor contains a list of attributes composing the group, separated by commas. If the attribute is shared, the Possible Values descriptor contains the name of the bus resource regular attribute that must be shared with another device. For type T attributes, the Possible Values descriptor contains the message numbers in a comma-separated list.

Descriptor	Description
Width	If the attribute is a regular attribute of type M for a bus memory address or of type O for a bus I/O address, the Width descriptor can be used to identify the amount in bytes of the bus memory or bus I/O space that must be allocated. Alternatively, the `Width` field can be set to a null string, which indicates that the amount of bus memory or bus I/O space is specified by a width attribute, that is, an attribute of type W. If the attribute is a regular attribute of type W, the Width descriptor contains the name of the bus memory address or bus I/O address attribute to which this attribute corresponds. The use of a type W attribute allows the amount of bus memory or bus I/O space to be configurable, whereas if the amount is specified in the bus memory address or bus I/O address attribute's Width descriptor, it is fixed at that value and cannot be customized. For all other attributes, a null string is used to fill in this field.
Attribute Type	Identifies the attribute type. Only one attribute type must be specified. The characters A, B, M, I, N, O, P, and W represent bus resources that are regular attributes. For regular attributes that are not bus resources, the following attribute types are defined: **L** Indicates the microcode file base name and the text from the label on the diskette containing the microcode file. Only device's with downloadable microcode have attributes of this type. The L attribute type is used by the **chkmcode** program to determine whether a device which is present has any version of its microcode installed. If none is installed, the user is prompted to insert the microcode diskette with the label identified by this attribute. The base name is stored in the `Default Value` field and is the portion of the microcode file name not consisting of the level and version numbers. The label text is stored in the `Possible Values` field. **T** Indicates message numbers corresponding to possible text descriptions of the device. These message numbers are within the catalog and set identified in the device's PdDv object. A single PdDv object can represent many device types. Normally, the message number in a device's PdDv object also identifies its text description. However, there are cases where a single PdDv object represents different device types. This happens when the parent device which detects them cannot distinguish between the types. For example, a single PdDv object is used for both the 120MB and 160MB Direct Attached Disk drives. For these devices, unique device descriptions can be assigned by setting the message number in the device's PdDv object to 0 and having a T attribute type, indicating the set of possible message numbers. The device's configure method determines the actual device type and creates a corresponding CuAt object indicating the message number of the correct text description. **R** Indicates any other regular attribute that is not a bus resource.

Descriptor	Description
	Z If the attribute name is `led`, than this indicates the LED number for the device. Normally, the LED number for a device is specified in the device's PdDv object. However, in cases where the PdDv object may be used to respresent multiple device types, unique LED numbers can be assigned to each device type by having a type **Z** attribute with an attribute name of `led`. In this case, the LED number in the PdDv object is set to 0. The device's configure method determines the actual LED number for the device, possibly by obtaining the value from the device, and creates a corresponding CuAt object indicating the LED number. The default value specified in the type **Z** PdAt object with the attribute name of `led` is the LED number to be used until the device's configure method has determined the LED number for the device.
	The following are the bus resources types for regular attributes:
	A Indicates DMA arbitration level.
	B Indicates a bus memory address which is not associated with DMA transfers.
	M Indicates a bus memory address to be used for DMA transfers.
	I Indicates bus interrupt level that can be shared with another device.
	N Indicates a bus interrupt level that cannot be shared with another device.
	O Indicates bus I/O address.
	P Indicates priority class.
	W Indicates an amount in bytes of bus memory or bus I/O space.
	For non-regular attributes, the following attribute types are defined:
	G Indicates a group.
	S Indicates a shared attribute.
Generic Attribute Flags	Identifies the flags that can apply to any regular attribute. Any combination (one, both, or none) of these flags is valid. This descriptor should be a null string for group and shared attributes. This descriptor is always set to a null string for type T attributes.
	These are the defined generic attribute flags:
	D Indicates a displayable attribute**.** The **lsattr** command displays only attributes with this flag.
	U Indicates an attribute whose value can be set by the user.

Descriptor	Description
Attribute Representation Flags	Indicates the representation of the regular attribute values. For group and shared attributes, which have no associated attribute representation, this descriptor is set to a null string. Either the **n** or **s** flag, both of which indicate value representation, must be specified. The **r**, **l**, and **m** flags indicate, respectively, a range, an enumerated list, and a multi-select value list, and are optional. If neither the **r** flag nor the **l** flag is specified, the **attrval** subroutine will not verify that the value falls within the range or the list. These are the defined attribute representation flags: **n** Indicates that the attribute value is numeric: either decimal, hex, or octal. **s** Indicates that the attribute value is a character string. **r** Indicates that the attribute value is a range of the form: *lowerlimit-upperlimit,increment value*. **l** Indicates that the attribute value is an enumerated list of values. **m** Indicates that multiple values can be assigned to this attribute. Multiple values for an attribute are represented as a comma separated list. **b** Indicates that value is a boolean type, and can only have 2 values. Typical values are `yes,no`, `true,false`, `on,off`, `disable,enable` or `0,1`. The attribute representation flags are always set to **nl** (numeric list) for type T attributes.
NLS Index	Identifies the message number in the NLS message catalog of the message containing a textual description of the attribute. Only displayable attributes, as identified by the Generic Attribute Flags descriptor, need an NLS message. If the attribute is not displayable, the NLS index can be set to a value of 0. The catalog file name and the set number associated with the message number are stored in the PdDv object class.

Related Information

Predefined Devices (PdDv) object class, Customized Attribute object class.

The **attrval** subroutine.

The **chdev** command, **lsattr** command, **mkdev** command.

Adapter-Specific Considerations for the Predefined Attribute (PdAt) Object Class .

Predefined Attribute Extended (PdAtXtd) Object Class

Description

The Predefined Attribute Extended (PdAtXtd) object class is used to supplement existing device's attributes represented in the Predefined Attribute (PdAt) object class with information that can be used by Device Management User Interface. The Web-based System Manager Device application is the first user interface application to take advantage of this object class.

Types of Attributes to represent in PdAtXtd

Not all existing device's attributes in PdAt need to be represented in the PdAtXtd object class. Non-displayable attributes (i.e with a null string in the 'generic' field of the PdAt object class) should not have a corresponding PdAtXtd entry, otherwise, it will become displayable.

The PdAtXtd object class can also be used to override the current value or possible values of an attribute.

Predefined Attribute Extended Object Class Descriptors

The Predefined Attribute Extended object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Required
ODM_CHAR	**uniquetype**	Unique Type	Yes
ODM_CHAR	**attribute**	Attribute Name	No
ODM_CHAR	**classification**	AttributeClassification	No
ODM_CHAR	**sequence**	Sequence number	No
ODM_VCHAR	**operation**	Operation Name	No
ODM_VCHAR	**operation_value**	Operation Value	No
ODM_VCHAR	**description**	Attribute Description	No
ODM_VCHAR	**list_cmd**	Command to list Attribute value	No
ODM_VCHAR	**list_values_cmd**	Command to list Attribute values	No
ODM_VCHAR	**change_cmd**	Command to change Attribute value	No
ODM_VCHAR	**help**	Help text	NO
ODM_VCHAR	**nls_values**	Translated Attribute values	No

These descriptors are described as follows:

Descriptor — **Description**

Unique Type — Identifies the class-subclass-type name of the device to which this attribute is associated. This descriptor is the same as the Unique Type descriptor in the PdAt object class.

Attribute Name — Identifies the device attribute. This is the name that can be passed to mkdev and chdev configuration commands and device methods in the attribute-name and attribute-value pairs.

Classification — Identifies the device attribute's classification. The followings characters are valid values:

- **B** Indicates a basic attribute.
- **A** Indicates an advanced attribute.
- **R** Indicates a required attribute.

Sequence — Identifies the number used to position the attribute in relation to others on a panel/menu. This field is identical to the 'id_seq_num' currently in the sm_cmd_opt (SMIT Dialog/Selector Command Option) object class.

Descriptor	Description

Operation	Identifies the type of operation associated with the unique device type. Operation and attribute name fields are mutually exclusive. The following operation names are used by Web-based System Manager Device application:
	
	assign_icon
	Indicates that an icon is to be assigned to the unique device type.
	
	add_*device type*
	Indicates that the unique device type can be manually added to the system via the Web-based System Manager Device Application's 'New' Device action. *device type* is a user chosen name that will identify the type or class of device that can be added via the Web-based System Manager Device Application. This name will be sorted in alphabetical order, therefore, to have all similar type or class of devices be grouped together in the Web-based System Manager device selection panel, choose the name accordingly. Example:
	`add_isa_tokenring`
	`add_isa_ethernet`
	`add_tty`
	
	will allow the selections for adding ISA adapters (token ring and ethernet) be together, but
	`add_tokenring_isa`
	`add_ethernet_isa`
	`add_tty`
	
	will cause the selection for adding tty to be inserted in between the two ISA adapters selections.
	
	move_*device type*
	Indicates that the unique device type can be moved to another location via the Web-based System Manager Device Application's **Move** action
	
	list_*parent*
	Indicates that the unique device type has a special method to obtain the list of parent devices that it can be connected to. The method must be listed in the **list_cmd** field.
	
	show_apply_option
	Indicates that a selection will appear on the device properties panel, to allow the user to apply change(s) to devices' properties immediately, or defer the change(s) until the next System Restart.

Operation Value	Identifies the value associated with the Operation field. For Web-based System Manager Device Application, when the operation is 'assign_icon', the value in operation_value will be the name of the icon associated with the unique device type. The icon name is the first extension of the icon file name under /usr/websm/codebase/images directory. When the operation is 'add_<device>', the operation_value field may contain the command used to make the device, if the 'mkdev' command cannot be used. However, Web-based System Manager Device Application will invoke the command stored in this field with the same arguments normally passed to the 'mkdev' command.
Description	Identifies the attribute's description. Web-based System Manager Device Application expects this field to be of the following format: message file,set id,msg id,default text

Descriptor	Description
List Cmd	Identifies the command to issue to override the attribute's current value, except when operation field is set, then it will be the command to issue to return information associated with the operation. For example: In the case of 'add_tty' operation, the list_cmd field contains the following value: `lsdev -P -c tty -s rs232 -Fdescription` The string returned from executing this command will be put on the Web-based System Manager device selection panel.
List Values Cmd	Identifies the command to issue in order to obtain the possible values of an attribute. The values returned will override the values field in the Predefined Attribute object class.

Change Cmd	Commands used to change the attribute value if 'chdev' cannot be used.
	Note: When commands (stored in <field>_cmd) are executed to obtain information for an attribute, Web-based System Manager Device Application will always pass the device name as an argument to the command. Therefore, it is essential that the command stored in these <field>_cmd, handle this fact. Otherwise, a script can be stored in these fields in the following manner:
	`list_cmd = "x()\n\`
	`{\n\`
	`<run some command>\n\`
	`}\n\`
	`x "`
	
	In the case of the change_cmd field, Web-based System Manager Device Application will also pass in the attribute=value pair after the first argument.

Help	Help text associated with the attribute. This could be of the form:
	`message file,set id,msg id,default text`
	
	OR
	
	`a numeric string equal to a SMIT identifier tag.`

Nls Values	Identifies the text associated with the attribute's values. These values will be displayed in place of the values stored in the Predefined Attribute object class. This field should be of the form: `message file,set id,msg id,default text` The ordering of values should match the ordering in the Predefined Attribute values field.

Adapter-Specific Considerations for the Predefined Attribute (PdAt) Object Class

Description

The various bus resources required by an adapter card are represented as attributes in the Predefined Attribute (PdAt) object class. If the currently assigned values differ from the default values, they are represented with other device attributes in the Customized Attribute (CuAt) object class. To assign bus resources, the Bus Configurator obtains the bus resource attributes for an adapter from both the PdAt and CuAt object classes. It also updates the CuAt object class, as necessary, to resolve any bus resource conflicts.

The following additional guidelines apply to bus resource attributes.

The Attribute Type descriptor must indicate the type of bus resource. The values are as follows:

Value	Description
A	Indicates a DMA arbitration level.
B	Indicates a bus memory address which is not associated with DMA transfers.
M	Indicates a bus memory address to be used for DMA transfers.
I	Indicates a bus interrupt level that can be shared with another device.
N	Indicates a bus interrupt level that cannot be shared with another device.
O	Indicates a bus I/O address.
P	Indicates an interrupt-priority class.
W	Indicates an amount in bytes of bus memory or bus I/O space.
G	Indicates a group.
S	Indicates an attribute that must be shared with another adapter.

For bus memory and bus I/O addresses, the amount of address space to be assigned must also be specified. This value can be specified by either the attribute's Width descriptor or by a separate type W attribute.

If the value is specified in the attribute's Width descriptor, it is fixed at that value and cannot be customized. If a separate type W attribute is used, the bus memory or bus I/O attribute's Width descriptor must be set to a null string. The type W attribute's Width descriptor must indicate the name of the bus memory or bus I/O attribute to which it applies.

Attribute types G and S are special-purpose types that the Bus Configurator recognizes. If an adapter has resources whose values cannot be assigned independently of each other, a Group attribute will identify them to the Bus Configurator. For example, an adapter card might have an interrupt level that depends on the bus memory address assigned. Suppose that interrupt level 3 must be used with bus memory address 0x1000000, while interrupt level 4 must be used with bus memory address 0x2000000. This relationship can be described using the Group attribute as discussed in ″Predefined Attribute (PdAt) Object Class″ .

Occasionally, all cards of a particular type or types must use the same bus resource when present in the system. This is especially true of interrupt levels. Although most adapter's resources can be assigned independently of other adapters, even those of the same type, it is not uncommon to find adapters that must share an attribute value. An adapter card having a bus resource that must be shared with another adapter needs a type S attribute to describe the relationship.

PdAt Descriptors for Type S Attributes

The PdAt descriptors for a type S attribute should be set as follows:

PdAt Descriptor Setting	Description
Unique Type	Indicates the unique type of the adapter.
Attribute Name	Specifies the name assigned to this attribute.
Default Value	Set to a null string.
Possible Values	Contains the name of the attribute that must be shared with another adapter or adapters.
Width	Set to a null string.
Attribute Type	Set to `S`.
Generic Attribute Flags	Set to a null string. This attribute must neither be displayed nor set by the user.
Attribute Representation Flags	Set to `sl`, indicating an enumerated list of strings, even though the list consists of only one item.
NLS Index	Set to `0` since the attribute is not displayable.

The type S attribute identifies a bus resource attribute that must be shared. The other adapters are identifiable by attributes of type S with the same attribute name. The attribute name for the type S attribute serves as a key to identify all the adapters.

For example, suppose an adapter with unique type `adapter/mca/X` must share its interrupt level with an adapter of unique type `adapter/mca/Y`. The following attributes describe such a relationship:

The Predefined Attribute object for `X`'s interrupt level:

- Attribute Name = `int_level`
- Default Value = `3`
- Possible Values = `2 - 9, 1`
- Width = `null string`
- Unique Type = `adapter/mca/X`
- Attribute Type = `I`
- Generic Attribute Flags = D (displayable, but cannot be set by user)
- Attribute Representation Flags = `nr`
- NLS Index = `12` (message number for text description)

The predefined attribute object describing `X`'s shared interrupt level:

- Unique Type = `adapter/mca/X`
- Attribute Name = `shared_intr`
- Default Value = `null string`
- Possible Values = `"int_level"`
- Width = `null string`
- Attribute Type = `S`
- Generic Attribute Flags = `null string`
- Attribute Representation Flags = `sl`
- NLS Index = `0`

The Predefined Attribute object for `Y`'s interrupt level:

- Unique Type = `adapter/mca/Y`
- Attribute Name = `interrupt`
- Default Value = `7`
- Possible Values = `2, 3, 4, 5, 7, 9`
- Width = `null string`
- Attribute Type = `I`
- Generic Attribute Flags = `D` (displayed, but cannot be set by user)
- Attribute Representation Flags = `nl`
- NLS Index = `6` (message number for text description).

The Predefined Attribute object describing `Y`'s *shared* interrupt level:

- Unique Type = `adapter/mca/Y`
- Attribute Name = `shared_intr`
- Default Value = null string
- Possible Values = `"interrupt"`
- Width = `null string`
- Attribute Type = `S`

- Generic Attribute Flags = `null string`
- Attribute Representation Flags = `sl`
- NLS Index = `0`

Note that the two adapters require different attributes to describe their interrupt levels. The attribute name is also different. However, their attributes describing what must be shared have the same name: `shared_intr`.

Adapter bus resource attributes except those of type W can be displayed but not set by the user. That is, the Generic Attribute Flags descriptor can either be a null string or the character `D`, but cannot be `U` or `DU`. The Bus Configurator has total control over the assignment of bus resources. These resources cannot be changed to user-supplied values by the Change method.

The Bus Configurator uses type W attributes to allocate bus memory address and bus I/O address attributes but never changes the value of a type W attribute. Attributes of type W can be set by users by setting the Generic Attribute flags descriptor to `DU`. This allows the Change method to change the type W attribute values to a user-supplied value.

The Bus Configurator does not use or modify any other attribute the adapter may have with attribute type R.

Related Information

Customized Attributes (CuAt) object class,Predefined Attribute (PdAt) object class.

Device Methods for Adapter Cards: Guidelines .

Writing a Change Method .

Understanding Interrupts, Understanding Direct Memory Access (DMA), Writing a Device Method in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Object Data Manager (ODM) Overview for Programmers in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

Device Configuration Subsystem Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Predefined Connection (PdCn) Object Class

Description

The Predefined Connection (PdCn) object class contains connection information for intermediate devices. This object class also includes predefined dependency information. For each connection location, there are one or more objects describing the subclasses of devices that can be connected. This information is useful, for example, in verifying whether a device instance to be defined and configured can be connected to a given device.

Descriptors

The Predefined Connection object class contains the following descriptors:

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	**uniquetype[UNIQUESIZE]**	Unique Type	Required
ODM_CHAR	**connkey[KEYSIZE]**	Connection Key	Required

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	**connwhere[LOCSIZE]**	Connection Location	Required

These fields are described as follows:

Field	Description
Unique Type	Identifies the intermediate device's class-subclass-type name. For a device with dependency information, this descriptor identifies the unique type of the device on which there is a dependency. This descriptor contains the same information as the Unique Type descriptor in the Predefined Devices (PdDv) object class.
Connection Key	Identifies a subclass of devices that can connect to the intermediate device at the specified location. For a device with dependency information, this descriptor serves to identify the device indicated by the Unique Type field to the devices that depend on it.
Connection Location	Identifies a specific location on the intermediate device where a child device can be connected. For a device with dependency information, this descriptor is not always required and consequently may be filled with a null string. The term *location* is used in a generic sense. For example, for a bus device the location can refer to a specific slot on the bus, with values `1`, `2`, `3`,.... For a multiport serial adapter device, the location can refer to a specific port on the adapter with values `0`, `1`,....

Related Information

Predefined Devices (PdDv) object class.

Predefined Devices (PdDv) Object Class

Description

The Predefined Devices (PdDv) object class contains entries for all device types currently on the system. It can also contain additional device types if the user has specifically installed certain packages that contain device support for devices that are not on the system. The term *devices* is used generally to mean both intermediate devices (for example, adapters) and terminal devices (for example, disks, printers, display terminals, and keyboards). Pseudo-devices (for example, pseudo terminals, logical volumes, and TCP/IP) are also included there. Pseudo-devices can either be intermediate or terminal devices.

Each device type, as determined by class-subclass-type information, is represented by an object in the PdDv object class. These objects contain basic information about the devices, such as device method names and instructions for accessing information contained in other object classes. The PdDv object class is referenced by the Customized Devices (CuDv) object class using a link that keys into the Unique Type descriptor. This descriptor is uniquely identified by the class-subclass-type information.

Typically, the Predefined database is consulted but never modified during system boot or run time, except when a new device is added to the Predefined database. In this case, the predefined information for the new device must be added into the Predefined database. However, any new predefined information for a new base device must be written to the boot file system to be effective. This is done with the **bosboot** command.

You build a Predefined Device object by defining the objects in a file in stanza format and then processing the file with the **odmadd** command or the **odm_add_obj** subroutine. See the **odmadd** command or the **odm_add_obj** subroutine for information on creating the input file and compiling the object definitions into objects.

Note: When coding an object in this object class, set unused empty strings to "" (two double-quotation marks with no separating space) and unused integer fields to 0 (zero).

Descriptors

Each Predefined Devices object corresponds to an instance of the PdDv object class. The descriptors for the Predefined Devices object class are as follows:

Predefined Devices

ODM Type	Descriptor Name	Description	Descriptor Status
ODM_CHAR	**type[TYPESIZE]**	Device Type	Required
ODM_CHAR	**class[CLASSIZE]**	Device Class	Required
ODM_CHAR	**subclass[CLASSIZE]**	Device Subclass	Required
ODM_CHAR	**prefix[PREFIXSIZE]**	Prefix Name	Required
ODM_CHAR	**devid[DEVIDSIZE]**	Device ID	Optional
ODM_SHORT	**base**	Base Device Flag	Required
ODM_SHORT	**has_vpd**	VPD Flag	Required
ODM_SHORT	**detectable**	Detectable/Non-detectable Flag	Required
ODM_SHORT	**chgstatus**	Change Status Flag	Required
ODM_SHORT	**bus_ext**	Bus Extender Flag	Required
ODM_SHORT	**inventory_only**	Inventory Only Flag	Required
ODM_SHORT	**fru**	FRU Flag	Required
ODM_SHORT	**led**	LED Value	Required
ODM_SHORT	**setno**	Set Number	Required
ODM_SHORT	**msgno**	Message Number	Required
ODM_VCHAR	**catalog[CATSIZE]**	Catalog File Name	Required
ODM_CHAR	**DvDr[DDNAMESIZE]**	Device Driver Name	Optional
ODM_METHOD	**Define**	Define Method	Required
ODM_METHOD	**Configure**	Configure Method	Required
ODM_METHOD	**Change**	Change Method	Required
ODM_METHOD	**Unconfigure**	Unconfigure Method	Optional*
ODM_METHOD	**Undefine**	Undefine Method	Optional*
ODM_METHOD	**Start**	Start Method	Optional
ODM_METHOD	**Stop**	Stop Method	Optional
ODM_CHAR	**uniquetype[UNIQUESIZE]**	Unique Type	Required

These descriptors are described as follows:

Descriptor
Device Type

Description
Specifies the product name or model number. For example, IBM 3812-2 Model 2 Page printer and IBM 4201 Proprinter II are two types of printer device types. Each device type supported by the system should have an entry in the PdDv object class.

Descriptor	Description
Device Class	Specifies the functional class name. A functional class is a group of device instances sharing the same high-level function. For example, `printer` is a functional class name representing all devices that generate hardcopy output.
Device Subclass	Identifies the device subclass associated with the device type. A device class can be partitioned into a set of device subclasses whose members share the same interface and typically are managed by the same device driver. For example, parallel and serial printers form two subclasses within the class of printer devices. The configuration process uses the subclass to determine valid parent-child connections. For example, an rs232 8-port adapter has information that indicates that each of its eight ports only supports devices whose subclass is rs232. Also, the subclass for one device class can be a subclass for a different device class. In other words, several device classes can have the same device subclass.
Prefix Name	Specifies the Assigned Prefix in the Customized database, which is used to derive the device instance name and **/dev** name. For example, `tty` is a Prefix Name assigned to the tty port device type. Names of tty port instances would then look like `tty0`, `tty1`, or `tty2`. The rules for generating device instance names are given in the Customized Devices object class under the Device Name descriptor.
Base Device Flag	A base device is any device that forms part of a minimal base system. During the first phase of system boot, a minimal base system is configured to permit access to the root volume group and hence to the root file system. This minimal base system can include, for example, the standard I/O diskette adapter and a SCSI hard drive. The Base Device flag is a bit mask representing the type of boot for which the device is considered a base device. The **bosboot** command uses this flag to determine what predefined device information to save in the boot file system. The **savebase** command uses this flag to determine what customized device information to save in the boot file system. Under certain conditions, the **cfgmgr** command also uses the Base Device flag to determine whether to configure a device.
VPD Flag	Specifies whether device instances belonging to the device type contain extractable vital product data (VPD). Certain devices contain VPD that can be retrieved from the device itself. A value of TRUE means that the device has extractable VPD, and a value of FALSE that it does not. These values are defined in the **/usr/include/sys/cfgdb.h** file.
Detectable/Nondetectable Flag	Specifies whether the device instance is detectable or nondetectable. A device whose presence and type can be electronically determined, once it is actually powered on and attached to the system, is said to be detectable. A value of TRUE means that the device is detectable, and a value of FALSE that it is not. These values are defined in the **/usr/include/sys/cfgdb.h** file.

Descriptor	Description
Change Status Flag	Indicates the initial value of the Change Status flag used in the Customized Devices (CuDv) object class. Refer to the corresponding descriptor in the CuDv object class for a complete description of this flag. A value of NEW means that the device is to be flagged as new, and a value of DONT_CARE means ″it is not important.″ These values are defined in the **/usr/include/sys/cfgdb.h** file.
Bus Extender Flag	Indicates that the device is a bus extender. The Bus Configurator uses the Bus Extender flag descriptor to determine whether it should directly invoke the device's Configure method. A value of TRUE means that the device is a bus extender, and a value of FALSE that it is not. These values are defined in the **/usr/include/sys/cfgdb.h** file. This flag is further described in ″Device Methods for Adapter Cards: Guidelines″ .
Inventory Only Flag	Distinguishes devices that are represented solely for their replacement algorithm from those that actually manage the system. There are several devices that are represented solely for inventory or diagnostic purposes. Racks, drawers, and planars represent such devices. A value of TRUE means that the device is used solely for inventory or diagnostic purposes, and a value of FALSE that it is not used solely for diagnostic or inventory purposes. These values are defined in the **/usr/include/sys/cfgdb.h** file
FRU Flag	Identifies the type of field replaceable unit (FRU) for the device. The three possible values for this field are: **NO_FRU** Indicates that there is no FRU (for pseudo-devices). **SELF_FRU** Indicates that the device is its own FRU. **PARENT_FRU** Indicates that the FRU is the parent. These values are defined in the **/usr/include/sys/cfgdb.h** file.
LED Value	Indicates the hexadecimal value displayed on the LEDs when the Configure method executes.
Catalog File Name	Identifies the file name of the NLS message catalog that contains all messages pertaining to this device. This includes the device description and its attribute descriptions. All NLS messages are identified by a catalog file name, set number, and message number.
Set Number	Identifies the set number that contains all the messages for this device in the specified NLS message catalog. This includes the device description and its attribute descriptions.
Message Number	Identifies the message number in the specified set of the NLS message catalog. The message corresponding to the message number contains the textual description of the device.

Descriptor	Description
Device Driver Name	Identifies the base name of the device driver associated with all device instances belonging to the device type. For example, a device driver name for a keyboard could be `ktsdd`. For the tape device driver, the name could be `tapedd`. The Device Driver Name descriptor can be passed as a parameter to the **loadext** routine to load the device driver, if the device driver is located in the **/usr/lib/drivers** directory. If the driver is located in a different directory, the full path must be appended in front of the Device Driver Name descriptor before passing it as a parameter to the **loadext** subroutine.
Define Method	Names the Define method associated with the device type. All Define method names start with the **def** prefix.
Configure Method	Names the Configure method associated with the device type. All Configure method names start with the **cfg** prefix.
Change Method	Names the Change method associated with the device type. All Change method names start with the **chg** prefix.
Unconfigure Method	Names the Unconfigure method associated with the device type. All Unconfigure method names start with the **ucfg** prefix. **Note:** The Optional* descriptor status indicates that this field is optional for those devices (for example, the bus) that are never unconfigured or undefined. For all other devices, this descriptor is required.
Undefine Method	Names the Undefine method associated with the device type. All Undefine method names start with the **und** prefix. **Note:** The Optional* descriptor status indicates that this field is optional for those devices (for example, the bus) that are never unconfigured or undefined. For all other devices, this descriptor is required.
Start Method	Names the Start method associated with the device type. All Start method names start with the **stt** prefix. The Start method is optional and only applies to devices that support the Stopped device state.
Stop Method	Names the Stop method associated with the device type. All Stop method names start with the **stp** prefix. The Stop method is optional and only applies to devices that support the Stopped device state.
Unique Type	A key that is referenced by the **PdDvLn** link in CuDv object class. The key is a concatenation of the Device Class, Device Subclass, and Device Type values with a **/** (slash) used as a separator. For example, for a class of `disk`, a subclass of `scsi`, and a type of `670mb`, the Unique Type is `disk/scsi/670mb`. This descriptor is needed so that a device instance's object in the CuDv object class can have a link to its corresponding PdDv object. Other object classes in both the Predefined and Customized databases also use the information contained in this descriptor.

Files

/usr/lib/drivers directory	Contains device drivers.

Related Information

Customized Devices (CuDv) object class.

The **loadext** subroutine, **odm_add_obj** subroutine.

The **odmadd** command.

Writing a Define Method , Writing a Configure Method , Writing a Change Method , Writing an Undefine Method , Writing an Unconfigure Method , Writing Optional Start and Stop Methods .

Adapter-Specific Considerations for the Predefined Devices (PdDv) Object Class

Description

The information to be populated into the Predefined Devices object class is described in the Predefined Devices (PdDv) Object Class. The following descriptors should be set as indicated:

Device Class	Set to `adapter`.
Device ID	Must identify the values that are obtained from the POS(0) and POS(1) registers on the adapter card. The format is `0xAABB`, where `AA` is the hexadecimal value obtained from POS(0), and `BB` the value from POS(1). This descriptor is used by the Bus Configurator to match up the physical device with its corresponding information in the Configuration database.
Bus Extender Flag	Usually set to FALSE, which indicates that the adapter card is not a bus extender. This descriptor is set to TRUE for a multi-adapter card requiring different sets of bus resources assigned to each adapter. The Standard I/O Planar is an example of such a card.

The Bus Configurator behaves slightly differently for cards that are bus extenders. Typically, it finds an adapter card and returns the name of the adapter to the Configuration Manager so that it can be configured.

However, for a bus extender, the Bus Configurator directly invokes the device's Configure method. The bus extender's Configure method defines the various adapters on the card as separate devices (each needing its own predefined information and device methods), and writes the names to standard output for the Bus Configurator to intercept. The Bus Configurator adds these names to the list of device names for which it is to assign bus resources.

An example of a type of adapter card that would be a bus extender is one which allows an expansion box with additional card slots to be connected to the system.

Related Information

Adapter-Specific Considerations for the PdAt Object Class .

Writing a Configure Method .

Predefined Devices (PdDv) object class.

Chapter 2. Communications Subsystem

ddclose Communications PDH Entry Point

Purpose

Frees up system resources used by the specified communications device until they are needed.

Syntax

#include <sys/device.h>
int ddclose (*devno*, *chan*) **dev_t** *devno*; **int** *chan*;

Parameters

devno Major and minor device numbers.
chan Channel number assigned by the device handler's **ddmpx** entry point.

Description

The **ddclose** entry point frees up system resources used by the specified communications device until they are needed again. Data retained in the receive queue, transmit queue, or status queue is purged. All buffers associated with this channel are freed. The **ddclose** entry point should be called once for each successfully issued **ddopen** entry point.

Before issuing a **ddclose** entry point, a **CIO_HALT** operation should be issued for each previously successful **CIO_START** operation on this channel.

Execution Environment

A **ddclose** entry point can be called from the process environment only.

Return Value

In general, communication device-handlers use the common return codes defined for entry points. However, device handlers for specific communication devices may return device-specific codes. The common return code for the **ddclose** entry point is the following:

ENXIO Indicates an attempt to close an unconfigured device.

Related Information

The **ddmpx** entry point, **ddopen** entry point.

The **CIO_HALT** ddioctl Communications PDH Operation, **CIO_START** ddioctl Communications PDH Operation.

dd_fastwrt Communications PDH Entry Point

Purpose

Allows kernel-mode users to transmit data.

Description

You use the **dd_fastwrt** entry point from a kernel-mode process to pass a write packet or string of packets to a PDH for transmission. To get the address of this entry point, you issue the **fp_ioctl** (**CIO_GET_FASTWRT**) kernel service.

The syntax and rules of usage are device-dependent and therefore not listed here. See the documentation on individual devices for more information. Some of the information that should be provided is:

- Number of packets allowed on a single fast write function call.
- Operational level from which the fast write function can be called.
- Syntax of the entry point.
- Trusted path usage. The device may not check every parameter.

When you call this entry point from a different adapter's receive interrupt level, you must ensure that the calling level is equal to or lower than the target adapter's operational level. This is the case when you forward packets from one port to another. To find out the operational level, see the documentation for the specific device.

Related Information

The **fp_ioctl** kernel service.

CIO_GET_FASTWRT ddioctl Communications PDH Operation

Purpose

Provides the parameters required to issue a kernel-mode fast-write call.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int
ddioctl (devno, op, parmptr,
devflag, chan, ext)
dev_t  devno;
int  op;
struct status_block * parmptr;
ulong  devflag;
int  chan,  ext;
```

Description

The **CIO_GET_FASTWRT** operation returns the parameters required to issue a kernel-mode fast write for a particular device. Only a kernel-mode process can issue this entry point and use the fast-write function. The parameters returned are located in the **cio_get_fastwrt** structure in the **/usr/include/sys/comio.h** file.

Note: This operation should not be called by user-mode processes.

Parameters

devno	Specifies major and minor device numbers.
op	Indicates the entry point for the **CIO_GET_FASTWRT** operation.
parmptr	Points to a **cio_get_fastwrt** structure. This structure is defined in the **/usr/include/sys/comio.h** file.
devflag	Indicates the **DKERNEL** flag. This flag must be set, indicating a call by a kernel-mode process.
chan	Specifies the channel number assigned by the device-handler **ddmpx** entry point.

ext Specifies the extended subroutine parameter. This parameter is device-dependent.

Execution Environment

A **CIO_GET_FASTWRT** operation can be called from the process environment only.

Return Values

In general, communication device handlers use the common codes defined for an operation. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **CIO_GET_FASTWRT** operation are:

ENXIO Indicates an attempt to use an unconfigured device.
EFAULT Indicates that the specified address is not valid.
EINVAL Indicates a parameter call that is not valid.
EPERM Indicates a call from a user-mode process is not valid.
EBUSY Indicates the maximum number of opens was exceeded.
ENODEV Indicates the device does not exist.

Related Information

The **ddioctl** device driver entry point in *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1*.

The **ddwrite** entry point, **dd_fastwrt** entry point.

CIO_GET_STAT ddioctl Communications PDH Operation

Purpose

Returns the next status block in a status queue to user-mode process.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int ddioctl
(devno, op, parmptr,
devflag, chan, ext)
dev_t  devno;
int  op;
struct status_block * parmptr;
ulong  devflag;
int  chan,
 ext;
```

Parameters

devno Specifies major and minor device numbers.
op Indicates the entry point for the **CIO_GET_STAT** operation.
parmptr Points to a **status_block** structure. This structure is defined in the **/usr/include/sys/comio.h** file.
devflag Specifies the **DKERNEL** flag. This flag must be clear, indicating a call by a user-mode process.
chan Specifies the channel number assigned by the device-handler **ddmpx** entry point.
ext Indicates device-dependent.

Description

Note: This entry point should not be called by kernel-mode processes.

The **CIO_GET_STAT** operation returns the next status block in the status queue to a user-mode process.

Execution Environment

A **CIO_GET_STAT** operation can be called from the process environment only.

Return Values

In general, communication device handlers use the common codes defined for an operation. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **CIO_GET_STAT** operation are the following:

ENXIO	Indicates an attempt to use an unconfigured device.
EFAULT	Indicates the specified address is not valid.
EINVAL	Indicates a parameter is not valid.
EACCES	Indicates a call from a kernel process is not valid.
EBUSY	Indicates the maximum number of opens was exceeded.
ENODEV	Indicates the device does not exist.

Related Information

The **ddioctl** device driver entry point, **ddmpx** entry point in *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1*.

CIO_HALT ddioctl Communications PDH Operation

Purpose

Removes the network ID of the calling process and cancels the results of the corresponding **CIO_START** operation.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int ddioctl
(devno, op, parmptr,
devflag, chan, ext)
dev_t  devno;
int  op;
struct session_blk * parmptr;
ulong  devflag;
int  chan,  ext;
```

Parameters

devno	Specifies major and minor device numbers.
op	Specifies the entry point for the **CIO_HALT** operation.
parmptr	Points to a **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file.
devflag	Specifies the **DKERNEL** flag. This flag is set by kernel-mode processes and cleared by calling user-mode processes.
chan	Specifies the channel number assigned by the device handler's **ddmpx** routine.

ext Indicates device-dependent.

Description

The **CIO_HALT** operation must be supported by each physical device handler in the communication I/O subsystem. This operation should be issued once for each successfully issued **CIO_START** operation. The **CIO_HALT** operation removes the caller's network ID and undoes all that was affected by the corresponding **CIO_START** operation.

The **CIO_HALT** operation returns immediately to the caller, before the operation completes. If the return indicates no error, the PDH builds a **CIO_HALT _DONE** status block upon completion. For kernel-mode processes, the status block is passed to the associated status function (specified at open time). For user-mode processes, the block is placed in the associated status or exception queue.

session_blk Parameter Block

For the **CIO_HALT** operation, the *ext* parameter can be a pointer to a **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Indicates the status of the port. This field may contain additional information about the completion of the **CIO_HALT** operation. Besides the status codes listed here, device-dependent codes can be returned: **CIO_OK** Indicates the operation was successful. **CIO_INV_CMD** Indicates an invalid command was issued. **CIO_NETID_INV** Indicates the network ID was not valid. The `status` field is used for specifying immediately detectable errors. If the status is **CIO_OK**, the **CIO_HALT _DONE** status block should be processed to determine whether the halt completed without errors.
`netid`	Contains the network ID to halt.

Execution Environment

A **CIO_HALT** operation can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an operation. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **CIO_HALT** operation are the following:

Return Code	Description
ENXIO	Indicates an attempt to use an unconfigured device.
EFAULT	Indicates an incorrect address was specified.
EINVAL	Indicates an incorrect parameter was specified.
EBUSY	Indicates the maximum number of opens was exceeded.
ENODEV	Indicates the device does not exist.

Related Information

The **ddioctl** device driver entry point in *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1*.

The **CIO_GET_STAT** ddioctl Communications PDH Operation, **CIO_START** ddioctl Communications PDH Operation.

CIO_QUERY ddioctl Communications PDH Operation

Purpose

Returns device statistics.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int ddioctl
(devno, op, parmptr,
devflag, chan, ext)
dev_t  devno;
int  op;
struct query_parms * parmptr;
ulong  devflag;
int  chan,  ext;
```

Parameters

devno	Specifies major and minor device numbers.
op	Indicates the entry point of the **CIO_QUERY** operation.
parmptr	Points to a **query_parms** structure. This structure is defined in the **/usr/include/sys/comio.h** file.
devflag	Specifies the **DKERNEL** flag. This flag is set by calling kernel-mode processes and cleared by calling user-mode processes.
chan	Specifies channel number assigned by the device handler's **ddmpx** entry point.
ext	Indicates device-dependent.

Description

The **CIO_QUERY** operation returns various statistics from the device. Counters are zeroed by the physical device handler when the device is configured. The data returned consists of two contiguous portions. The first portion contains counters to be collected and maintained by all device handlers in the communication I/O subsystem. The second portion consists of device-dependent counters and parameters.

query_parms Parameter Block

For the **CIO_QUERY** operation, the *paramptr* parameter points to a **query_parms** structure. This structure is located in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Contains additional information about the completion of the status block. Besides the status codes listed here, the following device-dependent codes can be returned: **CIO_OK** Indicates the operation was successful. **CIO_INV_CMD** Indicates a command was issued that is not valid.
`bufptr`	Points to the buffer where the statistic counters are to be copied.
`buflen`	Indicates the length of the buffer pointed to by the `bufptr` field.
`clearall`	When set to **CIO_QUERY_CLEAR**, the statistics counters are set to 0 upon return.

Execution Environment

A **CIO_QUERY** operation can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **CIO_QUERY** operation are the following:

Return Code	Description
ENXIO	Indicates an attempt to use unconfigured device.
EFAULT	Indicates an address was specified that is not valid.
EINVAL	Indicates a parameter is not valid.
EIO	Indicates an error has occurred.
ENOMEM	Indicates the operation was unable to allocate the required memory.
EBUSY	Indicates the maximum number of opens was exceeded.
ENODEV	Indicates the device does not exist.

Related Information

The **ddioctl** device driver entry point, **ddmpx** entry point in BkSym.TRKernel5;.

CIO_START ddioctl Communications PDH Operation

Purpose

Opens a communication session on a channel opened by a **ddopen** entry point.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int ddioctl (devno, op, parmptr, devflag, chan, ext)
dev_t  devno;
int  op;
struct session_blk * parmptr;
ulong  devflag;
int  chan,  ext;
```

Parameters

devno	Specifies major and minor device numbers.
op	Specifies the entry point for the **CIO_START** operation.
parmptr	Points to a **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file.
devflag	Specifies the **DKERNEL** flag. This flag is set by calling kernel-mode processes and cleared by calling user-mode processes.
chan	Specifies the channel number assigned by the device handler's **ddmpx** entry point.
ext	Indicates device-dependent.

Description

The **CIO_START** operation must be supported by each physical device handler (PDH) in the communication I/O subsystem. Its use varies from adapter to adapter. This operation opens a

communication session on a channel opened by a **ddopen** entry point. Once a channel is opened, multiple **CIO_START** operations can be issued. For each successful start, a corresponding **CIO_HALT** operation must be issued later.

The **CIO_START** operation requires only the *netid* input parameter. This parameter is registered for the session. At least one network ID must be registered for this session before the PDH successfully accepts a call to the **ddwrite** or **ddread** entry point on this session. If this start is the first issued for this port or adapter, the appropriate hardware initialization is performed. Time-consuming initialization activities, such as call connection, are also performed.

This call returns immediately to the caller before the asynchronous command completes. If the return indicates no error, the PDH builds a **CIO_START_DONE** status block upon completion. For kernel-mode processes, the status block is passed to the associated status function (specified at open time). For user-mode processes, the status block is placed in the associated status or exception queue.

The session_blk Parameter Block

For the **CIO_START** operation, the *ext* parameter may be a pointer to a **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Indicates the status of the port. This field may contain additional information about the completion of the **CIO_START** operation. Besides the status codes listed here, device-dependent codes can also be returned: **CIO_OK** Indicates the operation was successful. **CIO_INV_CMD** Indicates an issued command was not valid. **CIO_NETID_INV** Indicates the network ID was not valid. **CIO_NETID_DUP** Indicates the network ID was a duplicate of an existing ID already in use on the network. **CIO_NETID_FULL** Indicates the network table is full.
`netid`	Contains the network ID to register with the start.

Execution Environment

A **CIO_START** operation can be called from the process environment only.

Return Values

In general, communication device-handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **CIO_START** operation are the following:

Return Code	Description
ENXIO	Indicates an attempt to use an unconfigured device.
EFAULT	Indicates a specified address is not valid.
EINVAL	Indicates a parameter is not valid.
ENOSPC	Indicates the network ID table is full.
EADDRINUSE	Indicates a duplicate network ID.
EBUSY	Indicates the maximum number of opens was exceeded.
ENODEV	Indicates the device does not exist.

Related Information

The **ddioctl** device driver entry point in *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1*.

The **CIO_GET_FASTWRT** ddioctl Communications PDH Operation, **CIO_GET_STAT** ddioctl Communications PDH Operation, **CIO_HALT** ddioctl Communications PDH Operation.

The **ddread** entry point, **ddwrite** entry point.

---

ddopen (Kernel Mode) Communications PDH Entry Point

Purpose

Performs data structure allocation and initialization for a communications physical device handler (PDH).

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int ddopen (devno, devflag, chan, extptr)
dev_t  devno;
ulong  devflag;
int  chan;
struct kopen_ext * extptr;
```

Parameters for Kernel-Mode Processes

devno Specifies major and minor device numbers.

devflag Specifies the flag word with the following definitions:

DKERNEL
Set to call a kernel-mode process.

DNDELAY
When set, the PDH performs nonblocking writes for this channel. Otherwise, blocking writes are performed.

chan Specifies the channel number assigned by the device handler's **ddmpx** entry point.

extptr Points to the **kopen_ext** structure.

Description

The **ddopen** entry point performs data structure allocation and initialization. Hardware initialization and other time-consuming activities, such as call initialization, are not performed. This call is synchronous, which means it does not return until the **ddopen** entry point is complete.

kopen_ext Parameter Block

For a kernel-mode process, the *extptr* parameter points to a **kopen_ext** structure. This structure contains the following fields:

Field	Description
status	The status field may contain additional information about the completion of an open. Besides the status code listed here, the following device-dependent codes can also be returned: **CIO_OK** Indicates the operation was successful. **CIO_NOMBUF** Indicates the operation was unable to allocate **mbuf** structures. **CIO_BAD_RANGE** Indicates a specified address or parameter was not valid. **CIO_HARD_FAIL** Indicates a hardware failure has been detected.
rx_fn	Specifies the address of a kernel procedure. The PDH calls this procedure whenever there is a receive frame to be processed. The **rx_fn** procedure must have the following syntax: **#include </usr/include/sys/comio.h>** **void rx_fn (***open_id***,** *rd_ext_p***,** *mbufptr***)** **ulong** *open_id***;** **struct read_extension ***rd_ext_p**;** **struct mbuf ***mbufptr**;** *open_id* Identifies the instance of open. This parameter is passed to the PDH with the **ddopen** entry point. *rd_ext_p* Points to the read extension as defined in the **/usr/include/sys/comio.h** file. *mbufptr* Points to an **mbuf** structure containing received data. The kernel procedure calling the **ddopen** entry point is responsible for pinning the **rx_fn** kernel procedure before making the open call. It is the responsibility of code scheduled by the **rx_fn** procedure to free the **mbuf** chain.
tx_fn	Specifies the address of a kernel procedure. The PDH calls this procedure when the following sequence of events occurs: 1. The **DNDELAY** flag is set (determined by its setting in the last uiop->uio_fmode field). 2. The most recent **ddwrite** entry point for this channel returned an **EAGAIN** value. 3. Transmit queue for this channel now has room for a write. The **tx_fn** procedure must have the following syntax: **#include </usr/include/sys/comio.h>** **void tx_fn (***open_id***)** **ulong** *open_id***;** *open_id* Identifies the instance of open. This parameter is passed to the PDH with the **ddopen** call. The kernel procedure calling the **ddopen** entry point is responsible for pinning the **tx_fn** kernel procedure before making the call.

Field	Description
stat_fn	Specifies the address of a kernel procedure to be called by the PDH whenever a status block becomes available. This procedure must have the following syntax:

#include </usr/include/usr/comio.h>

void stat_fn (*open_id***,** *sblk_ptr***);**

ulong *open_id***;**

**struct status_block ***sblk_ptr*

open_id
Identifies the instance of open. This parameter is passed to the PDH with the **ddopen** entry point.

sblk_ptr
Points to a status block defined in the **/usr/include/sys/comio.h** file.

The kernel procedure calling the **ddopen** entry point is responsible for pinning the **stat_fn** kernel procedure before making the open call.

The **rx_fn**, **tx_fn**, and **stat_fn** procedures are made synchronously from the off-level portion of the PDH at high priority from the PDH. Therefore, the called kernel procedure must return quickly. Parameter blocks are passed by reference and are valid only for the call's duration. After a return from this call, the parameter block should not be accessed.

Execution Environment

A **ddopen** (kernel mode) entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **ddopen** entry point are the following:

Return Code	Description
EINVAL	Indicates a parameter is not valid.
EIO	Indicates an error has occurred. The status field contains the relevant exception code.
ENODEV	Indicates there is no such device.
EBUSY	Indicates the maximum number of opens was exceeded, or the device was opened in exclusive-use mode.
ENOMEM	Indicates the PDH was unable to allocate the space that it needed.
ENXIO	Indicates an attempt was made to open the PDH before it was configured.
ENOTREADY	Indicates the PDH is in the process of shutting down the adapter.

Related Information

The **CIO_GET_FASTWRT** ddioctl Communications PDH Operation, **ddclose** entry point, **ddopen** entry point for user-mode processes, **ddwrite** entry point.

The **ddmpx** entry point.

Status Blocks for Communication Device Handlers Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications Physical Device Handler Model Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

ddopen (User Mode) Communications PDH Entry Point

Purpose

Performs data structure allocation and initialization for a communications physical device handler (PDH).

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int ddopen (devno, devflag, chan, ext)
dev_t  devno;
ulong  devflag;
int  chan;
int  ext;
```

Parameters for User-Mode Processes

devno Specifies major and minor device numbers.

devflag Specifies the flag word with the following definitions:

DKERNEL
This flag must be clear, indicating call by a user-mode process.

DNDELAY
If this flag is set, the PDH performs nonblocking reads and writes for this channel. Otherwise, blocking reads and writes are performed for this channel.

chan Specifies the channel number assigned by the device handler's **ddmpx** entry point.

ext Indicates device-dependent.

Description

The **ddopen** entry point performs data structure allocation and initialization. Hardware initialization and other time-consuming activities such as call initialization are not performed. This call is synchronous and does not return until the open operation is complete.

Execution Environment

A **ddopen** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices can return device-specific codes. The common return codes for the **ddopen** entry point are:

Return Code	Description
EINVAL	Indicates a parameter is not valid.
ENODEV	Indicates there is no such device.
EBUSY	Indicates the maximum number of opens was exceeded.
ENOMEM	Indicates the PDH was unable to allocate needed space.
ENOTREADY	Indicates the PDH is in the process of shutting down the adapter.
ENXIO	Indicates an attempt was made to open the PDH before it was configured.

Related Information

The **ddclose** entry point, **ddopen** entry point for kernel-mode processes.

ddread Communications PDH Entry Point

Purpose

Returns a data message to a user-mode process.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int ddread (devno, uiop, chan, extptr)
dev_t  devno;
struct uio * uiop;
int  chan;
read_extension * extptr;
```

Parameters

devno	Specifies major and minor device numbers.
uiop	Points to a **uio** structure. For a calling user-mode process, the **uio** structure specifies the location and length of the caller's data area in which to transfer information.
chan	Specifies the channel number assigned by the device handler's **ddmpx** entry point.
extptr	Indicates null or points to the **read_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file.

Description

Note: The entry point should not to be called by a kernel-mode process.

The **ddread** entry point returns a data message to a user-mode process. This entry point may or may not block, depending on the setting of the **DNDELAY** flag. If a nonblocking read is issued and no data is available, the **ddread** entry point returns immediately with 0 (zero) bytes.

For this entry point, the *extptr* parameter points to an optional user-supplied **read_extension** structure. This structure contains the following fields:

Field	Description
`status`	Contains additional information about the completion of the **ddread** entry point. Besides the status codes listed here, device-dependent codes can be returned: **CIO_OK** Indicates the operation was successful. **CIO_BUF_OVFLW** Indicates the frame was too large to fit in the receive buffer. The PDH truncates the frame and places the result in the receive buffer.
`netid`	Specifies the network ID associated with the returned frame. If a **CIO_BUF_OVFLW** code was received, this field may be empty.
`sessid`	Specifies the session ID associated with the returned frame. If a **CIO_BUF_OVFLW** code was received, this field may be empty.

Execution Environment

A **ddread** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **ddread** entry point are the following:

Return Code	Description
ENXIO	Indicates an attempt to use an unconfigured device.
EINVAL	Indicates a parameter is not valid.
EIO	Indicates an error has occurred.
EACCES	Indicates a call from a kernel process is not valid.
EMSGSIZE	Indicates the frame was too large to fit into the receive buffer and that no *extptr* parameter was supplied to provide an alternate means of reporting this error with a status of **CIO_BUF_OVFLW**.
EINTR	Indicates a locking mode sleep was interrupted.
EFAULT	Indicates a supplied address is not valid.
EBIDEV	Indicates the specified device does not exist.

Related Information

The **CIO_GET_FASTWRT** ddioctl Communication PDH Operation, **CIO_START** ddioctl Communication PDH Operation.

The **ddmpx** entry point, **ddwrite** entry point.

The **uio** structure.

---

ddselect Communications PDH Entry Point

Purpose

Checks to see whether a specified event or events has occurred on the device.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int ddselect (devno, events, reventp, chan)
dev_t  devno;
ushort  events;
ushort * reventp;
int  chan;
```

Parameters

devno Specifies major and minor device numbers.

events Specifies conditions to check. The conditions are denoted by the bitwise OR of one or more of the following:

POLLIN
Check whether receive data is available.

POLLOUT
Check whether transmit available.

POLLPRI
Check whether status is available.

POLLSYNC
Check whether asynchronous notification is available.

reventp Points to the result of condition checks. A bitwise OR of the following conditions is returned:

POLLIN
Indicates receive data is available.

POLLOUT
Indicates transmit available.

POLLPRI
Indicates status is available.

chan Specifies the channel number assigned by the device handler's **ddmpx** entry point.

Description

Note: This entry point should not be called by a kernel-mode process.

The **ddselect** communications PDH entry point checks and returns the status of 1 or more conditions for a user-mode process. It works the same way the common **ddselect** device driver entry point does.

Execution Environment

A **ddselect** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **ddselect** entry point are the following:

Return Code	Description
ENXIO	Indicates an attempt to use an unconfigured device.
EINVAL	Indicates a specified argument is not valid.
EACCES	Indicates a call from a kernel process is not valid.
EBUSY	Indicates the maximum number of opens was exceeded.
ENODEV	Indicates the device does not exist.

Related Information

The **ddmpx** entry point.

ddwrite Communications PDH Entry Point

Purpose

Queues a message for transmission or blocks until the message can be queued.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int ddwrite (devno, uiop, chan, extptr)
dev_t  devno;
struct uio * uiop;
int  chan;
struct write_extension * extptr;
```

Parameters

devno	Specifies major and minor device numbers.
uiop	Points to a **uio** structure specifying the location and length of the caller's data.
chan	Specifies the channel number assigned by the device handler's **ddmpx** entry point.
extptr	Points to a **write_extension** structure. If the *extptr* parameter is null, then default values are assumed.

Description

The **ddwrite** entry point either queues a message for transmission or blocks until the message can be queued, depending upon the setting of the **DNDELAY** flag.

The **ddwrite** communications PDH entry point determines whether the data is in user or system space by looking at the `uiop->uio_segflg` field. If the data is in system space, then the `uiop->uio_iov->iov_base` field contains an **mbuf** pointer. The **mbuf** chain contains the data for transmission. The `uiop->uio_resid` field has a value of 4. If the data is in user space, the data is located in the same manner as for the **ddwrite** device driver entry point.

write_extension Parameter Block

For this entry point, the *extptr* parameter can point to a **write_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Indicates the status of the port. This field may contain additional information about the completion of the **ddwrite** entry point. Besides the status codes listed here, device-dependent codes can be returned: **CIO_OK** Indicates that the operation was successful. **CIO_NOMBUF** Indicates that the operation was unable to allocate **mbuf** structures.

Field	Description
flag	Contains a bitwise OR of one or more of the following: **CIO_NOFREE_MBUF** Requests that the physical device handler (PDH) not free the **mbuf** structure after transmission is complete. The default is bit clear (free the buffer). For a user-mode process, the PDH always frees the **mbuf** structure. **CIO_ACK_TX_DONE** Requests that, when done with this operation, the PDH acknowledge completion by building a **CIO_TX_DONE** status block. In addition, requests that the PDH either call the kernel status function or (for a user-mode process) place the status block in the status or exception queue. The default is bit clear (do not acknowledge transmit completion).
writid	Contains the write ID to be returned in the **CIO_TX_DONE** status block. This field is ignored if the user did not request transmit acknowledgment by setting **CIO_ACK_TX_DONE** status block in the flag field.
netid	Contains the network ID.

Execution Environment

A **ddwrite** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices can return device-specific codes. The common return codes for the **ddwrite** entry point are the following:

Return Code	Description
ENXIO	Indicates an attempt to use an unconfigured device.
EINVAL	Indicates a parameter that is not valid.
EAGAIN	Indicates the transmit queue is full and the **DNDELAY** flag is set. The command was not accepted.
EFAULT	Indicates a specified address is not valid.
EINTR	Indicates a blocking mode sleep was interrupted.
ENOMEM	Indicates the operation was unable to allocate the needed **mbuf** space.
ENOCONNECT	Indicates a connection was not established.
EBUSY	Indicates the maximum number of opens was exceeded.
ENODEV	Indicates the device does not exist.

Related Information

The **CIO_GET_FASTWRT** ddioctl Communications PDH Operation, **CIO_GET_STAT** ddioctl Communications PDH Operation, **CIO_START** ddioctl Communications PDH Operation.

The **ddmpx** entry point.

The **uio** structure.

---

ent_fastwrt Ethernet Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides a faster means for a kernel user to transmit data from the Ethernet device.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>
#include <sys/entuser.h>
#include <sys/mbuf.h>

int ent_fastwrt(devno, m)
int  devno;
struct mbuf * m;
```

Description

By using the **ent_fastwrt** entry point, a kernel-mode user can transmit data more quickly than through the normal write system call. The address of the **ent_fastwrt** entry point, along with the *devno* parameter, is given to a kernel-mode caller by way of the **CIO_GET_FASTWRT** entioctl operation.

The **ent_fastwrt** entry point works with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

Parameters

devno	Specifies major and minor device numbers.
m	Points to an **mbuf** structure containing the caller's data.

Execution Environment

The **ent_fastwrt** entry point can be called from the kernel process environment or the interrupt environment. If the **ent_fastwrt** function is called from the interrupt environment it is the responsibility of the caller to ensure that the interrupt level is **ENT_OFF_LEVEL**, as defined in the **/usr/include/sys/entuser.h** file, or a less-favored priority.

The **ent_fastwrt** entry point does not support a multiple-packet write. The `m_nextpkt` field in the **mbuf** structure is ignored by the device driver.

The **ent_fastwrt** entry point does not support a write extension. The mbufs are freed when the transmit is complete, and no transmit acknowledgement is sent to the caller. If these defaults are not appropriate, use the normal **entwrite** entry point.

The **entwrite** entry point assumes a trusted caller. The parameter checking done in the normal **entwrite** entry point is not done in the **ent_fastwrt** entry point. The caller should ensure such things as a valid *devno* parameter and a valid mbuf length.

Return Values

ENODEV	Indicates that a minor number is not valid.
EAGAIN	Indicates that the transmit queue is full.

Related Information

The **entwrite** entry point.

The **CIO_GET_FASTWRT** entioctl Ethernet Device Handler Operation.

entclose Ethernet Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Resets the Ethernet device to a known state and returns system resources to the system.

Syntax

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

int entclose (devno, chan, ext)
dev_t  devno;
int  chan,  ext;
```

Parameters

devno	Identifies major and minor device numbers.
chan	Specifies the channel number assigned by the **entmpx** entry point.
ext	Ignored by the Ethernet device handler.

Description

The **entclose** entry point closes the device. It is called when a user-mode caller issues a **close** subroutine. Before issuing the **entclose** entry point, the caller should have issued a **CIO_HALT** operation for each successfully issued **CIO_START** operation during the particular instance of the open.

The **entclose** entry point functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

Note: For each **entopen** entry point issued, there must be a corresponding **entclose** entry point.

If the caller has specified a multicast address, the caller first needs to issue the appropriate **entioctl** operation to remove all multicast addresses before issuing the **entclose** entry point.

Execution Environment

An **entclose** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **entclose** entry point are the following:

Return Code	Description
ENXIO	Indicates that the device is not configured.
EBUSY	Indicates that the maximum number of opens was exceeded.
ENODEV	Indicates that the specified device does not exist.

Related Information

The **CIO_START** entioctl Communications PDH Operation.

The **close** subroutine.

The **entmpx** entry point, **entopen** entry point.

---

entconfig Ethernet Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Initializes, terminates, and queries the vital product data (VPD) of the Ethernet device handler.

Syntax

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

int entconfig (devno, cmd, uiop)
dev_t  devno;
int  cmd;
struct uio * uiop;
```

Parameters

devno Specifies major and minor device numbers.

cmd Specifies which of the following functions this routine should perform:

CFG_INIT
Initializes device handler and internal data areas.

CFG_TERM
Terminates the device handler.

CFG_QVPD
Queries VPD.

uiop Points to a **uio** structure. The **uio** structure is defined in the **/usr/include/sys/uio.h** file.

Description

The **entconfig** entry point initializes, terminates, and queries the VPD of the Ethernet device handler.

The following are three possible **entconfig** operations:

The **entconfig** entry point functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

Operation	Description
CFG_INIT	Registers entry point of the Ethernet device handler by placing them into the device switch table for the major device number specified by the *devno* parameter. The **uio** structure contains the *iov_base* pointer, which points to the Ethernet device-dependent structure (DDS). The caller provides the **uio** structure. The structure is copied into an internal save area by the **init** function.

Operation	Description
CFG_TERM	If there are no outstanding opens, the following occurs: • The Ethernet device handler marks itself terminated and prevents subsequent opens. • All dynamically allocated areas are freed. • All Ethernet device handler entry points are removed from the device switch table.
CFG_QVPD	Returns the Ethernet VPD to the caller. The VPD is placed in the area specified by the caller in the **uio** structure.

Execution Environment

An **entconfig** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **entconfig** entry point are the following:

Return Code	Description
EINVAL	Indicates an address range or op code (common to all **entconfig** *cmd* operations) is not valid.
EBUSY	Indicates the device was already open in Diagnostic Mode and the open request was denied (issued for **CFG_TERM** and **CFG_INIT** operations).
EEXIST	Indicates the DDS structure already exists (**CFG_TERM** operation).
ENODEV	Indicates no such device exists (issued for all three operations).
EUNATCH	Indicates the protocol driver was not attached (issued for the **CFG_TERM** operation).
EFAULT	Indicates a specified address (common to the **CFG_QVPD** and **CFG_INIT** operations) is not valid.
EINVAL	Indicates a range or op code (common to all three operations) is not valid.
EACCES	Indicates permission was denied because the device was already open, or because there were outstanding opens that were unable to terminate (common to the **CFG_TERM** and **CFG_QVPD** operations).
ENOENT	Indicates no DDS to delete (common to the **CFG_TERM** and **CFG_QVPD** operations).
ENXIO	Indicates no such device exists or the maximum number of adapters was exceeded (common to all three operations).
EEXIST	Indicates the DDS structure already exists (common to **CFG_TERM** and **CFG_INIT** operations).
EFAULT	Indicates a specified address (issued for **CFG_TERM** and **CFG_INIT** operations) is not valid.
ENOMEM	Indicates insufficient memory (issued for the **CFG_INIT** operation).

Related Information

Device-Dependent Structure (DDS) Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

The **uio** structure.

---

entioctl Ethernet Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides various functions for controlling the Ethernet device.

Syntax

```
#include <sys/device.h>
#include <sys/devinfo.h>
#include <sys/ioctl.h>
#include <sys/comio.h>
#include <sys/entuser.h>

int entioctl (devno, cmd, arg, devflag, chan, ext)
dev_t  devno;
int  cmd,  arg;
ulong  devflag;
int  chan,  ext;
```

Parameters

devno	Specifies major and minor device numbers.
cmd	Specifies which operation to perform. The possible **entioctl** operation codes can be found in the **/usr/include/sys/ioctl.h** and **/usr/include/sys/comio.h** files.
arg	Specifies the address of the **entioctl** parameter block.
devflag	Specifies a parameter ignored by the Ethernet device handler.
chan	Specifies the channel number assigned by the **entmpx** routine.
ext	Specifies a parameter not used by the Ethernet device handler.

Description

The **entioctl** Ethernet device-handler entry point provides various functions for controlling the Ethernet device. Common **entioctl** operations are supplemented by **entioctl** operations available for diagnostic purposes.

The **entioctl** entry point functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

These are the common valid **entioctl** operations:

Operation	Description
CCC_GET_VPD	Returns vital product data (VPD) about the adapter.
CIO_GET_FASTWRT	Provides the parameters required to issue a fast write.
CIO_GET_STAT	Returns the current adapter and device handler status.
CIO_HALT	Halts a session and removes the registered network ID.
CIO_QUERY	Returns the current random access storage (RAS) counter values.
CIO_START	Starts a session and registers a network ID.
ENT_SET_MULTI	Sets or resets a multicast address.
IOCINFO	Returns I/O character information.

The following **entioctl** operations are for diagnostic purposes:

Operation	Description
CCC_TRCTBL	Returns the address of the internal device driver trace table.
CIO_MEM_ACC	Reads or writes data from or to selected adapter RAM addresses.
CIO_POS_ACC	Reads or writes a byte from or to a selected adapter POS register
CIO_REG_ACC	Reads or writes a byte from or to a selected adapter I/O register.

The following are DMA facilities operations:

Operation	Description
ENT_LOCK_DMA	Sets up (locks) a user buffer to DMA from or to the adapter.
ENT_UNLOCK_DMA	Clears (unlocks) a user buffer from DMA control.

Execution Environment

An **entioctl** entry point can be called from the process environment only.

Related Information

The **entmpx** entry point.

CCC_GET_VPD (Query Vital Product Data) entioctl Ethernet Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Returns vital product data (VPD) about the Ethernet adapter.

Description

The **CCC_GET_VPD** operation returns VPD about the Ethernet adapter. For this operation, the *arg* parameter points to the **vital_product_data** structure. This structure is defined in the **/usr/include/sys/ciouser.h** file and has the following fields:

Field	Description
status	Indicates the status of the VPD characters returned in the array of characters. Valid values for this status word are found in the **/usr/include/sys/ciouser.h** file: **VPD_NOT_READ** VPD data has not been obtained from the adapter. **VPD_NOT_AVAIL** VPD data is not available for this adapter. **VPD_INVALID** VPD data that was obtained is not valid. **VPD_VALID** VPD data was obtained and is valid.
`length`	Specifies the number of bytes that are valid in the VPD character array. This value can be 0, depending on the status returned.
`vpd[n]`	An array of characters that contain the adapter's VPD. The number of valid characters is determined by the length value.

The **CCC_GET_VPD** operation functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

Execution Environment

A **CCC_GET_VPD** operation can be called from the process environment only.

Return Values

The return codes for the **CCC_GET_VPD** operation are:

Return Code	Description
EFAULT	Indicates a specified address is not valid.
ENXIO	Indicates no such device exists.

Related Information

The **entioctl** entry point.

The Vital Product Data Structure (VPD) for the Ethernet Device Handler in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

CIO_GET_FASTWRT (Get Fast Write) entioctl Ethernet Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Returns the parameters required to issue an **ent_fastwrt** call.

Description

The **CIO_GET_FASTWRT** operation returns the parameters required to issue the kernel-mode fast write for the Ethernet adapter. The parameters are returned in the **cio_get_fastwrt** structure, which is defined in the **/usr/include/sys/comio.h** file. The *arg* pointer points to the **cio_get_fastwrt** structure, which contains the following fields:

Field	Description
`status`	Indicates the status condition that occurred; either **CIO_OK** or **CIO_INV_CMD**.
`fastwrt_fn`	Indicates the address of the fast write function.
`devno`	Specifies major and minor numbers of the device.

The **CIO_GET_FASTWRT** operation works with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the network adapter and network qualifications.

Execution Environment

The **CIO_GET_FASTWRT** operation can be called from a kernel-mode process only.

Return Values

EINVAL	Indicates that a parameter is not valid.
ENODEV	Indicates that a minor number is not valid.
ENXIO	Indicates an attempt to use an unconfigured device.
EPERM	Indicates the calling process is a user-mode process.
EBUSY	Indicates the maximum number of opens was exceeded.

Related Information

The **entwrite** entry point, **ent_fastwrt** entry point.

CIO_GET_STAT (Get Status) entioctl Ethernet Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Returns the current Ethernet adapter and device handler status.

Description

Note: Only user-mode callers can use the **CIO_GET_STAT** operation.

The **CIO_GET_STAT** operation returns the current Ethernet adapter and device handler status. The device handler fills in the parameter block with the appropriate information upon return. For this operation, the *arg* parameter points to a status block structure. This structure is defined in the **/usr/include/sys/comio.h** file.

The **CIO_GET_STAT** operation functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

Status Blocks for the Ethernet Device Handler

Status blocks are used to communicate status and exception information to user-mode processes.

User-mode processes receive a status block whenever they request a **CIO_GET_STAT** operation. A user-mode process can wait for the next available status block by issuing a **entselect** entry point with the specified **POLLPRI** event.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, **CIO_START_DONE**).

Ethernet-specific status blocks are:

- **CIO_START_DONE**
- **CIO_HALT_DONE**

The Ethernet device handler also returns the following general communications status blocks:

- **CIO_ASYNC_STATUS**
- **CIO_LOST_STATUS**
- **CIO_NULL_BLK**
- **CIO_TX_DONE**

CIO_START_DONE

On successful completion of the **CIO_START** entioctl operation, a status block having the following fields is provided:

Field	Status
`option[0]`	**CIO_OK**.
`option[1]`	The two high-order bytes contain the two high-order bytes of the network address. The two low-order bytes contain the middle two bytes of the network address.
`option[2]`	The two low-order bytes contain the two low-order bytes of the network address.

CIO_HALT_DONE

On successful completion of the **CIO_HALT** entioctl operation, a status block having the following fields is provided:

Field	Status
option[0]	**CIO_OK**
option[1]	Not used
option[2]	Not used

Execution Environment

A **CIO_GET_STAT** operation can be called from the process environment only.

Return Values

The return codes for the **CIO_GET_STAT** operation are:

Return Code	Description
EACCES	Indicates that permission was denied.
EBUSY	Indicates that the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
ENODEV	Indicates that no such device exists.
ENXIO	Indicates that an attempt was made to use an unconfigured device.

Related Information

The **entioctl** entry point.

CIO_HALT (Halt Device) entioctl Ethernet Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Ends a session with the Ethernet device handler.

Description

The **CIO_HALT** operation ends a session with the Ethernet device handler. The caller indicates the network ID to halt. This **CIO_HALT** operation corresponds with the **CIO_START** operation successfully issued with the specified network ID.

The **CIO_HALT** operation functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the network adapter specifications for more information on configuring the network adapter and network qualifications.

Data for the specified network ID is no longer received. Data already received for the specified network ID, before the **CIO_HALT** operation, is still passed up to a user-mode caller by the **entselect** and **entread** entry points. The **rx_fn** routine specified at open time passes data to a kernel-mode caller.

When a **CIO_HALT** operation has ended the last open session on a channel, the caller should then issue the **entclose** operation.

Note: If the caller has specified a multicast address, the caller first needs to issue the appropriate **entioctl** entry point to remove all the multicast addresses before issuing a **CIO_HALT** operation.

For a **CIO_HALT** operation, the *arg* parameter points to a **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	There are two possible returned status values: • **CIO_OK** • **CIO_NETID_INV**
`netid`	Specifies the network ID. When IEEE 802.3 Ethernet is being used, the network ID is placed in the least significant byte of the `netid` field.

Execution Environment

A **CIO_HALT** operation can be called from the process environment only.

Return Values

The return codes for the **CIO_HALT** operation are:

Return Code	Description
EINVAL	Indicates the specified network ID is not in the table.
EBUSY	Indicates the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
ENODEV	Indicates no such device exists.
ENXIO	Indicates an attempt to use an unconfigured device.

Related Information

The **CIO_START** entioctl Ethernet Device Handler Operation.

The **entioctl** entry point, **entread** entry point, **entselect** entry point.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

CIO_QUERY (Query Statistics) entioctl Ethernet Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Reads the counter values accumulated by the Ethernet device handler.

Description

The **CIO_QUERY** operation reads the counter values accumulated by the device handler. The counters are initialized to 0 (zero) by each **CIO_START** operation issued.

For the **CIO_QUERY** operation, the *arg* parameter points to a **query_parms** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Specifies the current status condition. This field accepts two possible status values: • **CIO_OK** • **COP_BUF_OVFLW**

Field	Description
buffptr	Specifies the address of a buffer where the returned statistics are to be placed.
bufflen	Specifies the length of the buffer.
clearall	When set to a value of **CIO_QUERY_CLEAR**, the counters are cleared upon completion of the call. This value is defined in the **/usr/include/sys/comio.h** file.

The **CIO_QUERY** operation specifies the device-specific information placed in the supplied buffer. The counter placed in the supplied buffer by this operation is the **ent_query_stats_t** structure, which is defined in the **/usr/include/sys/entuser.h** file.

The **CIO_QUERY** operation functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the network adapter specifications for more information on configuring the network adapter and network qualifications.

Execution Environment

A **CIO_QUERY** operation can be called from the process environment only.

Return Values

The return codes for the **CIO_QUERY** operation are:

Return Codes	Description
ENOMEM	Indicates insufficient memory.
EIO	Indicates the caller's buffer is too small.
EBUSY	Indicates the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
ENODEV	Indicates no such device exists.
ENXIO	Indicates an attempt to use an unconfigured device.

Related Information

The **entioctl** entry point, **entopen** entry point.

The **CIO_START** entioctl Ethernet Device Handler Operation.

CIO_START (Start Device) entioctl Ethernet Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Establishes a session with the Ethernet device handler.

Description

The **CIO_START** operation establishes a session with the Ethernet device handler. The caller notifies the device handler of the network ID that it will use. The caller can issue multiple **CIO_START** operations. For each successful start issued, there should be a corresponding **CIO_HALT** operation issued.

The **CIO_START** operation functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

If the **CIO_START** operation is the first issued, the device handler initializes and opens the Ethernet adapter. When the first **CIO_START** operation successfully completes, the adapter is ready to transmit and receive data. The Ethernet adapter can receive the following packet types:

- Packets matching the Ethernet adapter's burned-in address (or the address specified in the device-dependent structure (DDS))
- Broadcast packets
- Multicast packets
- Packets matching the network ID specified in the `netid` field

The Ethernet device handler allows a maximum of 32 network IDs. The network ID must correspond to the type field in a standard Ethernet packet or the destination service access point (DSAP) address in an IEEE 802.3 packet.

For the **CIO_START** operation, the *arg* parameter points to a **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	There are four possible returned status values: • **CIO_OK** • **CIO_NETID_FULL** • **CIO_NETID_DUP** • **CIO_HARD_FAIL**
`netid`	Specifies the network ID the caller uses on the network. When IEEE 802.3 Ethernet is being used, the network ID is placed in the least significant byte of the `netid` field. **Note:** The Ethernet device handler does not allow the caller to specify itself as the wildcard network ID.
`length`	This field is used to specify the number of valid bytes in the `netid` field for mixed Ethernet. Valid values are `1` or `2`.

After the **CIO_START** operation has successfully completed, the caller is free to issue any valid Ethernet command.

Note: The Ethernet device handler does not support indiscriminate addressing.

Execution Environment

A **CIO_START** operation can be called from the process environment only.

Return Values

The return codes for the **CIO_START** operation are the following:

Return Codes	Description
ENETUNREACH	Indicates the operation was unable to reach the network.
EBUSY	Indicates the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
ENODEV	Indicates no such device exists.
ENXIO	Indicates an attempt to use an unconfigured device.
ENOSPC	Indicates the **netid** table is full.
EADDRINUSE	Indicates a duplicate network ID.

Related Information

The **CIO_HALT** entioctl Ethernet Device Handler Operation.

The **entioctl** entry point.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

ENT_SET_MULTI (Set Multicast Address) entioctl Ethernet Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Sets the multicast address for the Ethernet device.

Description

The **ENT_SET_MULTI** operation sets the multicast address for the Ethernet device. For this operation, the *arg* parameter points to the **ent_set_multi_t** structure. This structure is defined in the **/usr/include/sys/entuser.h** file and contains the following fields:

Field	Description
`opcode`	Specifies whether to add or delete a multicast address. When this field is **ENT_ADD**, the multicast address is added to the multicast entry table. When this field is **ENT_DEL**, the multicast address is removed from the multicast entry table. Valid Ethernet types are defined in the **/usr/include/sys/entuser.h** file.
`multi_addr(6)`	Identifies the multicast address array where the `multi_addr(0)` field specifies the most significant byte and the `multi_addr(5)` field specifies the least significant byte.

The **ENT_SET_MULTI** operation functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

Note: The Ethernet device handler allows a maximum of 10 multicast addresses.

Execution Environment

An **ENT_SET_MULTI** operation can be called from the process environment only.

Return Values

The return codes for the **ENT_SET_MULTI** operation are:

Return Code	Description
EFAULT	Indicates the specified address is not valid.
EINVAL	Indicates the operation code is not valid.
ENOSPC	Indicates no space was left on the device. The multicast table is full.
ENOTREADY	Indicates the device was not ready. (The first **CIO_START** operation was not issued and not completed.)
EACCES	Indicates permission was denied. (The device was open in Diagnostic mode.)
EAFNOSUPPORT	Indicates the address family was not supported by protocol. (The multicast bit in the address was not set.)
ENXIO	Indicates no such device exists.

Related Information

The **CIO_START** entioctl Ethernet Device Handler Operation.

IOCINFO (Describe Device) entioctl Ethernet Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Returns a structure that describes the Ethernet device.

Description

The **IOCINFO** operation returns a structure that describes the Ethernet device. For this operation, the *arg* parameter points to the **ethernet** substructure, which is defined as part of the **devinfo** structure. This **devinfo** structure is located in the **/usr/include/sys/devinfo.h** file and contains the following fields:

Field	Description
`devtype`	Identifies the device type. The Ethernet type is **DD_NET_DH**. This label is defined in the **/usr/include/sys/devinfo.h** file.
`devsubtype`	Identifies the device subtype. The Ethernet subtype is **DD_EN**. This label can be found in the **/usr/include/sys/devinfo.h** file.
`broad_wrap`	Indicates the adapter's ability to receive its own packets. A value of 1 indicates that the adapter can receive its own packets. A value of 0 indicates that the adapter cannot receive its own packets. For this adapter, a value of 0 is returned.
`rdto`	Specifies the receive data transfer offset. This value indicates an offset (in bytes) into the data area of the receive page-sized **mbuf** structure. The device handler places received data in this buffer.
`haddr`	Identifies the 6-byte unique Ethernet adapter address. This address is the burned-in address that is readable from the adapter's vital product data (VPD). The most significant byte of the address is placed in the `haddr(0)` field. The least significant byte is placed in the address specified by the `haddr(5)` field.
`net_addr`	Identifies the 6-byte unique Ethernet adapter address currently being used by the Ethernet adapter card. This address is either the burned-in address (readable from the VPD) or the alternate address that can be used to configure the adapter. The most significant byte of the address is placed in the address specified by the `net_addr(0)` field. The least significant byte is placed in the address specified by `net_addr(5)` field.

The **IOCINFO** operation functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

The parameter block is filled in with the appropriate values upon return.

Execution Environment

An **IOCINFO** operation can be called from the process environment only.

Return Values

The return codes for the **IOCINFO** operation are:

Return Code	Description
EFAULT	Indicates a specified address is not valid.
EINVAL	Indicates an operation code is not valid.
ENXIO	Indicates no such device exists.

Related Information

The **entioctl** entry point.

entmpx Ethernet Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Allocates and deallocates a channel for an Ethernet device handler.

Syntax

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

int entmpx (devno, chanp, channame)
dev_t  devno;
int * chanp;
char * channame;
```

Parameters

devno Specifies the major and minor device numbers.

chanp Contains the channel ID passed as a reference parameter. If the *channame* parameter is null, this parameter is the channel ID to be deallocated. Otherwise, the *chanp* parameter is set to the ID of the allocated channel.

channame Points to the remaining path name describing the channel to allocate. The *channame* parameter accepts the following values:

null Deallocates the channel.

Pointer to a null string
Allows a normal open sequence of the Ethernet device on the channel ID generated by the **entmpx** entry point.

Pointer to a "D"
Allows the Ethernet device to be opened in Diagnostic mode on the channel ID generated by the **entmpx** entry point.

Description

The **entmpx** entry point allocates and deallocates a channel for an Ethernet device handler. This entry point is not called directly by a user. The kernel calls the **entmpx** entry point in response to an open or close request.

The **entmpx** entry point functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Note: If the Ethernet device has been successfully opened, any subsequent Diagnostic mode open requests is unsuccessful. If the device has been successfully opened in Diagnostic mode, all subsequent open requests is unsuccessful.

Execution Environment

An **entmpx** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **entmpx** entry point are the following:

Return Code	Description
EBUSY	Indicates the maximum number of opens was exceeded.
ENOMSG	No message of desired type.
ENODEV	Indicates the specified device does not exist.
ENXIO	Indicates the device is not configured.

Related Information

The **entopen** entry point.

---

entopen Ethernet Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Initializes the Ethernet device handler and allocates the required system resources.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>
#include <sys/entuser.h>

int entopen (devno, devflag, chan, ext)
dev_t  devno;
ulong  devflag;
int  chan,  ext;
```

Parameters

devno	Specifies major and minor device numbers for both kernel- and user-mode entry pointers.
devflag	Specifies the **DKERNEL** flag, which must be set for a kernel-mode entry pointer. This flag cannot be set for user-mode entry pointers.
chan	Specifies the channel number assigned by the **entmpx** routine for both kernel- and user-mode entry pointers.
ext	Points to a **kopen_ext** structure. This structure is defined in the **/usr/include/sys/comio.h** file. This parameter is valid only for kernel-mode users; it is null for user-mode users.

Description

The **entopen** entry point prepares the Ethernet device for transmitting and receiving data. It is called when a user-mode entry pointer issues an **open**, **openx**, or **creat** subroutine. After the **entopen** entry point has successfully completed, the entry pointer must issue a **CIO_START** operation before using the Ethernet device handler. The device handler is then opened for reading and writing data.

The **entopen** entry point functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

Execution Environment

An **entopen** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **entopen** entry point are the following:

Return Code	Description
EINVAL	Indicates a range or op code that is not valid, or that the device is not in diagnostic mode.
ENOMEM	Indicates insufficient memory.
ENOTREADY	Indicates that the device was not ready. The first **CIO_START** operation was not issued and hence not completed.
ENXIO	Indicates that no such device exists. (The maximum number of adapters was exceeded.)

Related Information

The **entclose** entry point, **entmpx** entry point.

The **open**, **openx**, or **create** subroutine.

The **CIO_START** entioctl Ethernet Device Handler Operation.

---

entread Ethernet Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means of receiving data from the Ethernet device handler.

Syntax

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

int entread (devno, uiop, chan, ext)
dev_t  devno;
struct uio * uiop;
int  chan,  ext;
```

Parameters

devno	Specifies major and minor device numbers.
uiop	Points to a **uio** structure. This structure is defined in the **/usr/include/sys/uio.h** file.
chan	Specifies the channel number assigned by the **entmpx** routine.
ext	Can specify the address of the **entread** parameter block. If the *ext* parameter is null, then no parameter block is specified.

Description

Note: The **entread** entry point should only be called by user-mode callers.

The **entread** entry point provides the means of receiving data from the Ethernet device handler. When a user-mode caller issues a **read**, **readx**, **readv**, or **readvx** subroutine, the kernel calls the **entread** entry point.

When the **entread** entry point is called, the file system fills in the **uio** structure fields with the appropriate values. In addition, the device handler copies the data into the buffer specified by the caller.

For the **entread** entry point, the *ext* parameter may point to the **read_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following field:

Field	Description
status	Contains one of the following status codes: • **CIO_OK** • **CIO_BUF_OVRFLW** • **CIO_NOT_STARTED**

The **entread** entry point functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

Execution Environment

An **entread** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **entread** entry point are the following:

Return Code	Description
EACCES	Indicates permission was denied because the device was already open. Diagnostic mode open request denied.
EFAULT	Indicates a specified address is not valid.
EINTR	Indicates an interrupted system call.
EIO	Indicates an I/O error.
EMSGSIZE	Indicates the data returned was too large for the buffer.
EBUSY	Indicates the maximum number of opens was exceeded.
ENODEV	Indicates the specified device does not exist.
ENOCONNECT	Indicates no connection was established.
ENXIO	Indicates an attempt to use an unconfigured device.

Related Information

The **entmpx** entry point.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

entselect Ethernet Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Determines whether a specified event has occurred on the Ethernet device.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>

int entselect (devno, events, reventp, chan)
dev_t  devno;
ushort  events;
ushort * reventp;
int  chan;
```

Parameters

devno	Specifies major and minor device numbers.
events	Identifies the events to check.
reventp	Returned events pointer passed by reference. This pointer is used by the **entselect** entry point to indicate which of the selected events are true when the call occurs.
chan	Specifies the channel number assigned by the **entmpx** entry point.

Description

Note: Only user-mode callers should use the **entselect** entry point.

The **entselect** entry point determines if a specified event has occurred on the Ethernet device. This entry point must be called with the **select** or **poll** subroutine.

When the Ethernet device handler is in a state in which the specified event cannot be satisfied (for example, an adapter failure), then the **entselect** entry point sets the returned event flags to 1. This prevents the **select** or **poll** subroutine from waiting indefinitely.

The **entselect** entry point functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

An **entselect** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **entselect** entry point are the following:

Return Code	Description
EACCES	Indicates permission was denied because the device had not been initialized. Indicates that the Diagnostic mode open request was denied. Indicates permission was denied because the call is from a kernel-mode process.
ENXIO	Indicates there was no such device. (Maximum number of adapters was exceeded.)
EBUSY	Indicates the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
ENODEV	Indicates no such device exists.

Related Information

The **CIO_GET_FASTWRT** ddioctl Communications PDH Operation, the **entmpx** entry point.

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

---

entwrite Ethernet Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for transmitting data from the Ethernet device.

Syntax

```
#include <sys/device.h>
#include <sys/uio.h>
#include <sys/comio.h>
#include <sys/entuser.h>

int entwrite (devno, uiop, chan, ext)
dev_t  devno;
struct uio * uiop;
int  chan,  ext;
```

Parameters

devno	Specifies major and minor device numbers.
uiop	Points to a **uio** structure that provides variables to control the data transfer operation. This **uio** structure is defined in the **/usr/include/sys/uio.h** file.
chan	Specifies the channel number assigned by the **entmpx** entry point.
ext	Specifies the address of the **entwrite** parameter block. If the *ext* parameter is null, then no parameter block is specified.

Description

The **entwrite** entry point provides the means for transmitting data for the Ethernet device. The kernel calls it when a user-mode caller issues a **write**, **writex**, **writev**, or **writevx** subroutine.

For a user-mode caller, the file system fills in the **uio** structure variables with the appropriate values. A kernel-mode caller must fill in the **uio** structure in the same manner as the general **ddwrite** entry point.

For the **entwrite** entry point, the *ext* parameter is a pointer to a **write_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Identifies the status of the write operation. This field is in the **write_extension** structure and accepts the following values: • **CIO_OK** • **CIO_TX_FULL**
`write_id`	For a user-mode caller, the `write_id` field is returned to the caller by the **CIO_GET_STAT** operation if the **ACK_TX_DONE** option is selected. For a kernel-mode caller, the `write_id` field is returned to the caller by the **stat_fn** function that was provided at open time.

The **entwrite** entry point functions with an Ethernet High-Performance LAN adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the network adapter and network qualifications.

Execution Environment

An **entwrite** entry point can be called from the process environment only.

Return Values

In general, communication device handlers use the common return codes defined for an entry point. However, device handlers for specific communication devices may return device-specific codes. The common return codes for the **entwrite** entry point are the following:

Return Code	Description
EAGAIN	Indicates the transmit queue is full.
EFAULT	Indicates a specified address is not valid.
EINTR	Indicates an interrupted system call.
EINVAL	Indicates an address range or op code is not valid.
ENOCONNECT	Indicates no connection was established.
ENOMEM	Indicates insufficient memory.
EBUSY	Indicates the maximum number of opens was exceeded.
ENODEV	Indicates the specified device does not exist.
ENXIO	Indicates an attempt to use an unconfigured device.

Related Information

The **entmpx** entry point, **entread** entry point, **ent_fastwrt** entry point.

The **CIO_GET_FASTWRT** ddioctl Communications PDH Operation.

The **write**, **writex**, **writev**, or **writevx** subroutine.

The **uio** structure.

---

mpclose Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Resets the Multiprotocol Quad Port (MPQP) adapter to a known state and returns system resources back to the system on the last close for that adapter.

Syntax

```
int mpclose (devno, chan, ext)
dev_t devno;
int chan, ext;
```

Parameters

devno	Specifies major and minor device numbers.
chan	Specifies the channel number assigned by the **mpmpx** entry point.
ext	Ignored by the MPQP device handler.

Description

The **mpclose** entry point routine resets the MPQP adapter to a known state and returns system resources to the system on the last close for that adapter. The port no longer accepts **mpread**, **mpwrite**, or **mpioctl** operation requests. The **mpclose** entry point is called in user mode by issuing a **close** system call. The **mpclose** entry point is invoked in response to an **fp_close** kernel service.

On an **mpclose** entry point, the MPQP device handler does the following:

- Frees all internal data areas for the corresponding **mpopen** entry point.
- Purges receive data queued for this **mpopen** entry point.

On the last **mpclose** entry point for a particular adapter, the MPQP device handler also does the following:

- Frees its interrupt level to the system.
- Frees the DMA channel.
- Disables adapter interrupts.
- Sets all internal data elements to their default settings.

The **mpclose** entry point closes the device. For each **mpopen** entry point issued, there must be a corresponding **mpclose** entry point.

Before issuing the **mpclose** entry point, the caller should issue a **CIO_HALT** operation for each **CIO_START** operation issued during that particular instance of open. If a close request is received without a preceding **CIO_HALT** operation, the functions of the halt are performed. A close request without a preceding **CIO_HALT** operation occurs only during abnormal termination of the port.

The **mpclose** entry point functions with a 4-port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **mpclose** entry point can be called from the process environment only.

Return Values

The common return codes for the **mpclose** entry point are:

Return Code	Description
ECHRNG	Indicates the channel number is too large.
ENXIO	Indicates the port initialization was unsuccessful. This code could also indicate that the registration of the interrupt was unsuccessful.
ECHRNG	Indicates the channel number is out of range (too high).

Related Information

The **mpconfig** entry point, **mpioctl** entry point, **mpmpx** entry point, **mpopen** entry point, **mpread** entry point, **mpselect** entry point, **mpwrite** entry point.

The **CIO_HALT** mpioctl MPQP Device Handler Operation, **CIO_START** mpioctl MPQP Device Handler Operation.

The **close** system call.

The **fp_close** kernel service.

mpconfig Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides functions for initializing and terminating the Multiprotocol Quad Port (MPQP) device handler and adapter.

Syntax

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

int mpconfig (devno, cmd, uiop)
dev_t devno;
int cmd;
struct uio *uiop;
```

Parameters

devno Specifies major and minor device numbers.

cmd Specifies the function to be performed by this routine. There are two possible functions:

CFG_INIT
Initializes device handler and internal data areas.

CFG_TERM
Terminates the device handler.

uiop Points to a **uio** structure. The **uio** structure is defined in the **/usr/include/sys/uio.h** file.

Description

The **mpconfig** entry point provides functions for initializing and terminating the MPQP device handler and adapter. It is invoked through the **/usr/include/sys/config** device driver at device configuration time. This entry point supports the following operations:

- **CFG_INIT**
- **CFG_TERM**

The **mpconfig** entry point functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **mpconfig** entry point can be called from the process environment only.

Related Information

The **mpclose** entry point, **mpioctl** entry point, **mpmpx** entry point, **mpopen** entry point, **mpread** entry point, **mpselect** entry point, **mpwrite** entry point.

The **ddconfig** routine.

MPQP Device Handler Interface Overview.

Communications I/O Subsystem: Programming Introduction.

mpioctl Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides various functions for controlling the Multiprotocol Quad Port (MPQP) adapter.

Syntax

```
#include <sys/devinfo.h>
#include <sys/ioctl.h>
#include <sys/comio.h>
#include <sys/mpqp.h>

int mpioctl
(devno, cmd, extptr, devflag, chan, ext)
dev_t devno;
int cmd, extptr;
ulong devflag;
int chan, ext;
```

Parameters

devno	Specifies major and minor device numbers.
cmd	Identifies the operation to be performed.
extptr	Specifies an address of the parameter block.
devflag	Allows **mpioctl** calls to inherit properties that were specified at open time. The MPQP device handler inspects the **DNDELAY** flag for ioctl calls. Kernel-mode data link control (DLC) sets the **DKERNAL** flag that must be set for an **mpopen** call.
chan	Specifies the channel number assigned by the **mpmpx** entry point.
ext	Not used by MPQP device handler.

Description

The **mpioctl** entry point provides various functions for controlling the MPQP adapter. There are 16 valid **mpioctl** operations, including:

Operation	Description
CIO_GET_STATUS	Gets the status of the current MPQP adapter and device handler.
CIO_HALT	Ends a session with the MPQP device handler.
CIO_START	Initiates a session with the MPQP device handler.
CIO_QUERY	Reads the counter values accumulated by the MPQP device handler.
MP_CHG_PARMS	Permits the DLC to change certain profile parameters after the MPQP device has been started.
MP_START_AR	Puts the MPQP port into Autoresponse mode.
MP_STOP_AR	Permits the MPQP port to exit Autoresponse mode.

The **mpioctl** entry point functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

The possible **mpioctl** operation codes can be found in the **/usr/include/sys/ioctl.h**, **/usr/include/sys/comio.h**, and **/usr/include/sys/mpqp.h** files.

Execution Environment

The **mpioctl** entry point can be called from the process environment only.

Return Values

The common return codes for the **mpioctl** entry point are:

Return Code	Description
ENOMEM	Indicates the no memory buffers (mbufs) or mbuf clusters are available.
ENXIO	Indicates the adapter number is out of range.

Related Information

The **mpclose** entry point, **mpconfig** entry point, **mpmpx** entry point,, **mpopen** entry point, **mpread** entry point, **mpselect** entry point, **mpwrite** entry point.

The **CIO_GET_STAT** mpioctl MPQP Device Handler Operation, **CIO_HALT** mpioctl MPQP Device Handler Operation, **CIO_QUERY** mpioctl MPQP Device Handler Operation, **CIO_START** mpioctl MPQP Device Handler Operation, **MP_CHG_PARMS** mpioctl MPQP Device Handler Operation, **MP_START_AR** and **MP_STOP_AR** mpioctl MPQP Device Handler Operations.

CIO_GET_STAT (Get Status) mpioctl MPQP Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Gets the status of the current Multiprotocol Quad Port (MPQP) adapter and device handler.

Description

Note: Only user-mode processes can use the **CIO_GET_STAT** operation.

The **CIO_GET_STAT** operation gets the status of the current MPQP adapter and device handler. For the MPQP device handler, both solicited and unsolicited status can be returned.

Solicited status is status information that is returned as a completion status to a particular operation. The **CIO_START**, **CIO_HALT**, and **mpwrite** operations all have solicited status returned. However, for many asynchronous events common to wide-area networks, these are considered unsolicited status. The asynchronous events are divided into three classes:

- Hard failures
- Soft failures
- Informational (or status-related) messages

The **CIO_GET_STAT** operation functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Status Blocks for the Multiprotocol Device Handler

For the **CIO_GET_STAT** operation, the *extptr* parameter points to a **status_block** structure. When returned, the device handler fills this structure with the appropriate information. The **status_block** structure is defined in the **/usr/include/sys/comio.h** file and returns one of seven possible status conditions:

Status blocks are used to communicate status and exception information to user-mode processes.

User-mode processes receive a status block whenever they request a **CIO_GET_STAT** operation. A user-mode process can wait for the next available status block by issuing a **mpselect** entry point with the specified **POLLPRI** event.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, **CIO_START_DONE**). Seven possible MPQP status blocks exist:

- **CIO_ASYNC_STATUS**
- **CIO_HALT_DONE**
- **CIO_START_DONE**
- **CIO_TX_DONE**
- **MP_END_OF_AUTO_RESP**
- **MP_RDY_FOR_MAN_DIAL**
- **MP_THRESH_EXC**

CIO_ASYNC_STATUS Status Block

Asynchronous status notifies the data link control of asynchronous events such as network and adapter failures.

Code	CIO_ASYNC_STATUS
option[0]	Can be one of the following:
	MP_DSR_DROPPED, MP_RCV_TIMEOUT, MP_RELOAD_CMPL, MP_RESET_CMPL, MP_X21_CLEAR
option[1]	Not used
option[2]	Not used
option[3]	Not used

Note: The **MP_RELOAD_C** and **MPLMP_RESET_CMPL** values are for diagnostic use only.

CIO_HALT_DONE Status Block

The **CIO_HALT** operation ends a session with the MPQP device handler. On a successfully completed Halt Device operation, the following status block is provided:

Code	CIO_HALT_DONE
option[0]	**CIO_OK**
option[1]	**MP_FORCED_HALT or MP_NORMAL_HALT**
option[2]	**MP_NETWORK_FAILURE or MP_HW_FAILURE**

A *forced halt* is a halt completed successfully in terms of the data link control is concerned, but terminates forcefully because of either an adapter error or a network error. This is significant for X.21 or other switched networks where customers can be charged if the call does not disconnect properly.

Note: When using the X.21 physical interface, X.21 centralized multiport (multidrop) operation on a leased-circuit public data network is not supported.

CIO_START_DONE Status Block

On a successfully completed **CIO_START** operation, the following status block is provided:

Code	CIO_START_DONE
option[0]	**CIO_OK**
option[1]	Network ID

Code	CIO_START_DONE
option[2]	Not used
option[3]	Not used

On an unsuccessful Start Device **CIO_START** mpioctl operation, the following status block is provided:

Code	CIO_START_DONE
option[0]	Can be one of the following: **MP_ADAP_NOT_FUNC** Adapter not functional **MP_TX_FAILSAFE_TIMEOUT** Transmit command did not complete. **MP_DSR_ON_TIMEOUT** DSR failed to come on. **MP_DSR_ALRDY_ON** DSR already on for a switched line. **MP_X21_RETRIES_EXC** X.21 retries exceeded. Call not completed. **MP_X21_TIMEOUT** X.21 timer expired. **MP_X21_CLEAR** Unexpected clear received from the DCE.
option[1]	If the option[0] field is set to **MP_X21_TIMEOUT**, the option[1] field contains the specific X.21 timer that expired.
option[2]	Not used.
option[3]	Not used.

CIO_TX_DONE Status Block

On completion of a multiprotocol transmit, the following status block is provided:

Code	CIO_TX_DONE
option[0]	Can be one of the following: **CIO_OK** **MP_TX_UNDERRUN** **MP_X21_CLEAR** **MP_TX_FAILSAFE_TIMEOUT** The transmit command did not complete. **MP_TX_ABORT** Transmit aborted due to CIO_HALT operation.
option[1]	Identifies the write_id field supplied by the caller in the write command if **TX_ACK** was selected.
option[2]	Points to the buffer with transmit data.
option[3]	Not used.

MP_END_OF_AUTO_RESP Status Block

The **MP_STOP_AR** mpioctl operation has completed. The adapter is in Normal Receive mode. All receive data is routed to the data link control.

Code	MP_END_OF_AUTO_RESP
option[0]	**CIO_OK**
option[1]	Not used
option[2]	Not used
option[3]	Not used

MP_RDY_FOR_MAN_DIAL Status Block
The manual dial switched line is ready for dialing. The start operation is pending the call completion.

Code	MP_RDY_FOR_MAN_DIAL
option[0]	**CIO_OK**
option[1]	Not used
option[2]	Not used
option[3]	Not used

MP_THRESH_EXC Status Block
A threshold for one of the counters defined in the start profile has reached its threshold.

Code	MP_THRESH_EXC
option[0]	Indicates the expired threshold.
	The following values are returned to indicate the threshold that was exceeded: **MP_TOTAL_TX_ERR, MP_TOTAL_RX_ERR, MP_TX_PERCENT, MP_RX_PERCENT**
option[1]	Not used.
option[2]	Not used.
option[3]	Not used.

Execution Environment
The **CIO_GET_STAT** operation can be called from the process environment only.

Return Values
The return codes for the **CIO_GET_STAT** operation are:

Return Code	Description
ENOMEM	Indicates no mbufs or mbuf clusters are available.
ENXIO	Indicates the adapter number is out of range.

Related Information
The **CIO_HALT** mpioctl MPQP Device Handler Operation, **CIO_QUERY** mpioctl MPQP Device Handler Operation, **CIO_START** mpioctl MPQP Device Handler Operation, **MP_CHG_PARMS** mpioctl MPQP Device Handler Operation, **MP_START_AR** and **MP_STOP_AR** mpioctl MPQP Device Handler Operations.

The **mpioctl** entry point, **mpwrite** entry point.

CIO_HALT (Halt Device) mpioctl MPQP Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Ends a session with the Multiprotocol Quad Port (MPQP) device handler and terminates the connection to the MPQP link.

Description

The **CIO_HALT** operation terminates a session with the MPQP device handler. The caller specifies which network ID to halt. The **CIO_HALT** operation removes the network ID from the network ID table and disconnects the physical link. A **CIO_HALT** operation must be issued for each **CIO_START** operation that completed successfully.

Data received for the specified network ID before the **CIO_HALT** operation is called can be retrieved by the caller using the **mpselect** and **mpread** entry points.

If the **CIO_HALT** operation terminates abnormally, the status is returned either asynchronously or as part of the **CIO_HALT_DONE**. Whatever the case, the **CIO_GET_STAT** operation is used to get information about the error. When a halt is terminated abnormally (for example, due to network failure), the following occurs:

- The link is terminated.
- The drivers and receivers are disabled for the indicated port.
- The port can no longer transmit or receive data.

No recovery procedure is required by the caller; however, logging the error is required.

Errors are reported on halt operations because the user could continue to be charged for connect time if the network does not recognize the halt. This error status permits a network application to be notified about an abnormal link disconnection and then take corrective action, if necessary.

The **CIO_HALT** operation functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Parameter Block

For the MPQP **CIO_HALT** operation, the *extptr* parameter points to a **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Specifies the status of the port. This field is set for immediately detectable errors. Possible values for the status filed are: • **CIO_OK** • **CIO_NETID_INV** If the calling process does not wish to sleep while the halt is in progress, the **DNDELAY** option can be used. In either case, the status of the halt is retrieved using the **CIO_GET_STATUS** operation and a **CIO_HALT_DONE** status block is returned. The **CIO_HALT_DONE** status block should be used as an indication of completion.
`netid`	Contains the network ID the caller wishes to halt. The network ID is placed in the least significant byte of the `netid` field.

Execution Environment

The **CIO_HALT** operation can be called from the process environment only.

Return Values

The **CIO_HALT** operation returns common communications return values. In addition, the following MPQP specific errors may be returned:

Error	Description
EBUSY	Indicates the device is not started or is not in a data transfer state.
ENOMEM	Indicates there are no mbufs or mbuf clusters available.
ENXIO	Indicates the adapter number is out of range.

Files

/usr/include/sys/comio.h	Contains the **session_blk** structure definition.

Related Information

The **mpread** entry point, **mpselect** entry point.

The **CIO_GET_STAT** mpioctl MPQP Device Handler Operation, **CIO_QUERY** mpioctl MPQP Device Handler Operation, **CIO_START** mpioctl MPQP Device Handler Operation, **MP_CHG_PARMS** mpioctl MPQP Device Handler Operation, **MP_START_AR** and **MP_STOP_AR** mpioctl MPQP Device Handler Operations.

Status Blocks for the Multiprotocol Device Handler.

CIO_QUERY (Query Statistics) mpioctl MPQP Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means to read counter values accumulated by the Multiprotocol Quad Port (MPQP) device handler.

Description

The **CIO_QUERY** operation reads the counter values accumulated by the MPQP device handler. The counters are initialized to 0 (zero) by the first **mpopen** entry point operation.

The **CIO_QUERY** operation returns the Reliability/Availability/Serviceability field of the define device structure (DDS).

The **CIO_QUERY** operation functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

The t_query_parms Parameter Block

For this operation, the *extptr* parameter points to an **t_query_parms** structure. This structure is defined in the **/usr/include/sys/mpqp.h** file and has the following fields:

Field	Description
status	Contains additional information about the completion of the status block. Device-dependent codes may also be returned.
CIO_OK	Indicates that the operation was successful.
bufptr	Specifies the address of a buffer where the returned statistics are to be placed.
buflen	Specifies the length of the buffer; it should be at least 45 words long (unsigned long).
reserve	Reserved for use in future releases.

Statistics Logged for MPQP Ports

The following statistics are logged for each MPQP port.

- Bytes transmitted
- Bytes received
- Frames transmitted
- Frames received
- Receive errors
- Transmission errors
- DMA buffer not large enough or not allocated
- Autoresponse transmission fail-safe time out
- Autoresponse received time out
- CTS time out
- CTS dropped during transmit
- DSR time out
- DSR dropped
- DSR on before DTR on a switched line
- X.21 call-progress signal (CPS)
- X.21 unrecognized CPS
- X.21 invalid CPS
- DCE clear during call establishment
- DCE clear during data phase
- X.21 T1-T5 time outs
- X.21 invalid DCE-provided information (DPI)

Note: When using the X.21 physical interface, X.21 centralized multiport (multidrop) operation on a leased-circuit public data network is not supported.

Execution Environment

The **CIO_QUERY** operation can be called from the process environment only.

Return Values

EFAULT	Indicates a specified address is not valid.
EINVAL	Indicates a parameter is not valid.
EIO	Indicates an error has occurred.
ENOMEM	Indicates the operation was unable to allocate the required memory.
ENXIO	Indicates an attempt to use unconfigured device.

Related Information

The **mpioctl** entry point, **mpopen** entry point.

The **CIO_GET_STAT** mpioctl MPQP Device Handler Operation, **CIO_HALT** mpioctl MPQP Device Handler Operation, **CIO_START** mpioctl MPQP Device Handler Operation, **MP_CHG_PARMS** mpioctl MPQP Device Handler Operation, **MP_START_AR** and **MP_STOP_AR** mpioctl MPQP Device Handler Operations.

CIO_START (Start Device) mpioctl MPQP Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Starts a session with the Multiprotocol Quad Port (MPQP) device handler.

Description

The **CIO_START** operation registers a network ID in the network ID table and establishes the physical connection with the MPQP device. Once this start operation completes successfully, the port is ready to transmit and receive data.

Note: The **CIO_START** operation defines the protocol- and configuration-specific attributes of the selected port. All bits that are not defined must be set to 0 (zero).

For the MPQP **CIO_START** operation, the *extptr* parameter points to a **t_start_dev** structure. This structure contains pointers to the **session_blk** structure.

The **session_blk** structure contains the `netid` and `status` fields. The **t_start_dev** device-dependent information for an MPQP device follows the session block. All of these structures can be found in the **/usr/include/sys/mpqp.h** file.

The **CIO_START** operation functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

t_start_dev Fields

The **t_start_dev** structure contains the following fields:

Field	Description
phys_link	Indicates the physical link protocol. Only one type of physical link is valid at a time. The six supported values are: **Physical Link Type** **PL_232D** EIA-232D **PL_422A** EIA-422A **PL_V35** V.35 **PL_X21** X.21 **PL_SMART_MODEM** Hayes EIA232-D autodial protocol **PL_V25** V.25bis EIA-422A autodial protocol **Note:** When using the X.21 physical interface, X.21 centralized multiport (multidrop) operation on a leased-circuit public data network is not supported. If the phys_link field is **PL_SMART_MODEM** or **PL_V25**, the dial_proto and dial_flags fields are applicable. Otherwise, these fields are ignored. If no dial protocol or flags are supplied when the **PL_SMART_MODEM** or **PL_V25** link is selected, defaults are used. The defaults for the dial phase for a **PL_SMART_MODEM** link is an asynchronous protocol at 2400 baud with even parity, 7 bits per character, and 1 stop bit. A **PL_V25** link has the same defaults.
dial_proto	Identifies the autodial protocol, which communicates with the modem to send information such as dial sequence or register settings. Most modems use an asynchronous protocol during the connect phase of call establishment. If no value is set, the default mode is asynchronous. **Note:** The dial_proto field is ignored if the physical link is not an autodial protocol.
data_proto	Identifies the possible data protocol selections during the data transfer phase. The data_flags field has different meanings depending on what protocol is selected. The data_proto field accepts the following values: **DATA_PRO_BSC** Indicates a bisync protocol. **DATA_PRO_SDLC_FDX** Indicates receivers enabled during transmit. **DATA_PRO_SDLC_HDX** Indicates receivers disabled during transmit.

Field	Description
modem_flags	Establishes modem characteristics. This field accepts the following values: **MF_AUTO** Indicates that the call is to be answered or dialed automatically. **MF_CALL** Indicates an outgoing call (switched only). **MF_CDSTL_OFF** Indicates that the data terminal ready (DTR) should be enabled without waiting for ring indicate (RI) to connect data set to line. **MF_CDSTL_ON** Enables DTR after RI occurs. If the data set ready (DSR) goes active prior to RI, DTR is enabled and RI is ignored. **MF_DRS_ON** Enables date rate selected (DRS). **MF_DRS_OFF** Disables DRS (full speed). This is the default. **MF_LEASED** Indicates a leased telephone circuit. **MF_LISTEN** Indicates an incoming call (switched only). **MF_MANUAL** Indicates that the operator answers or dials the call manually. **MF_SWITCHED** Indicates a switched telephone circuit. **Note:** The **MF_DRS_ON** and **MF_DRS_OFF** modem characteristics are not currently supported. The default is full speed.
poll_addr	Identifies the address-compare value for a Binary Synchronous Communication (BSC) polling frame or an Synchronous Data Link Control (SDLC) frame. If using BSC, a value for the selection address must also be provided or the address-compare is not enabled. If a frame is received that does not match the poll address (or select address for BSC), the frame is not passed to the system.
select_addr	Specifies a valid select address for BSC only.
modem_int_mask	Reserved. This value must be 0.

Field	Description
baud_rate	Specifies the baud rate for transmit and receive clock. This field is used for date terminal equipment (DTE) clocking only (that is, when the modem does not supply the clock). Acceptable baud rates range from 150 baud to a maximum speed of 38400 baud. If this field contains a value that does not match one of the following choices, the next lowest baud rate is used: • 38400 • 19200 • 9600 • 4800 • 2400 • 2000 • 1200 • 1050 • 600 • 300 A value of 0 indicates the port is to be externally clocked (that is, use modem clocking). The on-board baud rate generator is limited to a speed of 38400. All higher baud rates up to the maximum of 64000 bits *must* be accomplished with modem clocking. For RS232, the adapter drives a clock signal on the DTE clock. Most modems provide their own clocking. If the physical link is set to the **PL_SMART_MODEM** or **PL_V25** link, the baud rate is the speed of the dial sequence and modem clocking is used for data transfer.
rcv_timeout	Indicates the period of time, expressed in 100-msec units (0.10 sec), used for setting the receive timer. The MPQP device driver starts the receive timer whenever the **CIO_START** operation completes and a final transmit occurs. If a receive occurs that is not a receive final frame, the timer is restarted. The timer is stopped when the receive final occurs. If the timer expires before a receive occurs, an error is reported to the logical link control (LLC) protocol. After the **CIO_START** operation completes, the receive time out value can be changed by the **MP_CHANGE_PARAMS** operation. A value of zero indicates that a receive timer should not be activated. Final frames in SDLC are all frames with the poll or final bit set. In BSC, all frames are final frames except intermediate text block (ITB) frames.
rcv_data_offset	Reserved
dial_data_length	Specifies the length of the dial data. Dial data for Hayes-style dial data can be up to 256 bytes.

Flag Fields for Autodial Protocols

Flag fields in the **t_start_dev** structure take different values depending on the type of autodial protocol selected.

Data Flags for the BSC Autodial Protocol

If BSC is selected in the data_proto field, either ASCII or EBCDIC character sets can be used. Control characters are stripped automatically on reception. Data link escape (DLE) characters are automatically inserted and deleted in transparent mode. If BSC Address Check mode is selected, values for both poll and select addresses must be supplied. Odd parity is used if ASCII is selected.

The following are the default values:

- EBCDIC.

- Do not restart the receive timer.
- Do not check addresses.
- RTS controlled.

The data flags for the BSC autodial protocol are:

Data Flag	Description
DATA_FLG_RST_TMR	Reset receive timer.
DATA_FLG_ADDR_CHK	Address-compare select. This causes frames to be filtered by the hardware based on address. Only frames with matching addresses are sent to the system.
DATA_FLG_BSC_ASC	ASCII BSC select.
DATA_FLG_C_CARR_ON	Continuous carrier (RTS always on).
DATA_FLG_C_CARR_OFF	RTS-disabled between transmits (default).

Dial Flags for ASC Protocols

If ASC and the parity enable bit is set, the value for parity select is honored. A value of 0 equals even parity. A value of 1 equals odd parity. If parity enable is set to 0, no parity type is enforced. The following are acceptable ASC dial flags:

ASC Dial Flag	Description
DIAL_FLG_PAR_EN	Enable parity.
DIAL_FLG_PAR_ODD	Odd parity.
DIAL_FLG_STOP_0	0 stop bits.
DIAL_FLG_STOP_1	1 stop bit.
DIAL_FLG_STOP_15	1.5 stop bits.
DIAL_FLG_STOP_2	2 stop bits.
DIAL_FLG_CHAR_5	5 bits per character.
DIAL_FLG_CHAR_6	6 bits per character.
DIAL_FLG_CHAR_7	7 bits per character.
DIAL_FLG_CHAR_8	8 bits per character.
DIAL_FLG_C_CARR_ON	Continuous carrier (RTS always on).
DIAL_FLG_C_CARR_OFF	RTS disabled between transmits (default).
DIAL_FLG_TX_NO_CTS	Transmit without waiting for clear to send (CTS).
DIAL_FLG_TX_CTS	Wait for CTS to transmit (default).

Data Flags for the SDLC Protocol

For the Synchronous Data Link Control (SDLC) protocol, the flag for NRZ or NRZI must match the data-encoding method that is used by the remote DTE. If SDLC Address Check mode is selected, the poll address byte must also be specified. The receive timer (RT) is started whenever a final block is transmitted. If RT is set to 1, the receive timer is restarted after expiration. If RT is set to 0, the receive timer is not restarted after expiration. The receive timer value is specified by the 16-bit `rcv_time out` field. The following are the acceptable SDLC data flags:

SLDC Data Flag	Description
DATA_FLG_NRZI	NRZI select (default is NRZ).
DATA_FLG_ADDR_CHK	Address-compare select.
DATA_FLG_RST_TMR	Restart receive timer.
DIAL_FLG_C_CARR_ON	Continuous carrier (RTS always on).
DIAL_FLG_C_CARR_OFF	RTS disabled between transmits (default).

t_auto_data Fields

The **t_auto_data** structure contains the following fields that specify aspects of the X.21 call progress signal retry and logging data format:

Field	Description
`len`	Length of autodial to be sent to the modem.
`sig[]`	Signals to be sent to the modem data in the form of an array of characters.
`connect_timer`	Time-out value. This value is specified in 0.1 seconds. The adapter should wait for call to complete before reporting a connection failure to the DLC. The default is 30 seconds if no value is set.
`v25b_tx_timer`	Time-out value. This value is specified in 0.1 seconds of delay after driving DTR and before sending dial data in V.25bis modem protocol. If no value is set, a default value of 0.1 seconds is used.

t_x21_data Fields

The **t_x21_data** structure contains the following fields that specify aspects of the X.21 call progress signal retry and logging data format:

Field	Description
`selection signal length`	Contains the length of the data held in the selection-signals portion of the buffer in bytes. Values from 0 to 256 are valid.
`selection signals`	The selection signals are allocated 256 bytes each. Items are stored in the International Alphabet 5 (IA5) format.
`retry_cnt`	Indicates how many attempts at outgoing call establishment must fail before the adapter software returns an error to the driver for the **CIO_START** operation. Values between 0 and 255 are allowed. This is a 1-byte field.
`retry_delay`	Contains the number of 100-msec (0.1 sec) intervals to wait between successive call retries. This is a 2-byte field.
`cps_group`	There are nine characters-per-second (cps) groups. Each cps group can generate a driver interrupt after a configurable number of errors are detected. Optionally, this interrupt can cause an X.21 network transaction to notify network error-logging monitors of excessive error rates. The **netlog** bit definitions determine which signals in each group are considered countable.

Retry and Netlog Groups

Specify the `retry` and `netlog` fields for a cps-particular group. The bits definitions are as follows:

- In the `retry` field, a 1-bit (On) indicates that retries are enabled for this signal.
- In logging fields, a set bit indicates that errors of this type should be counted in the cumulative group error statistics. Eventually, these statistics can generate interrupts to the driver.

Call progress signals are divided into groups of 10; for example, cps 43 is group 4, signal 3. To indicate retries for cps 43, the value for signal 3 should be ORed into the retry unsigned short for group 4. Possible values for retry groups are the following:

- **CG_SIG_0**
- **CG_SIG_1**
- **CG_SIG_2**
- **CG_SIG_3**
- **CG_SIG_4**
- **CG_SIG_5**

- **CG_SIG_6**
- **CG_SIG_7**
- **CG_SIG_8**
- **CG_SIG_9**

t_err_threshold Fields

The **t_err_threshold** structure describes the format for defining thresholds for transmit and receive errors. Counters track the total number of transmit and receive errors. Individual counters track certain types of errors. Thresholds can be set for individual errors, total errors, or a percentage of transmit and receive errors from all frames received.

When a counter reaches its threshold value, a status block is returned by the driver. The status block indicates the type of error counter that reached its threshold. If multiple thresholds are reached at the same time, the first expired threshold in the list is reported as having expired and its counter is reset to 0. The user can issue a **CIO_QUERY** operation call to retrieve the values of all counters.

If no thresholding is desired, the threshold should be set to 0. A value of 0 indicates that LLC should not be notified of an error at any time. To indicate that the LLC should be notified of every occurrence of an error, the threshold should be set to 1.

The **t_err_threshold** structure contains the following fields:

Field	Description
`tx_err_thresh`	Specifies the threshold for all transmit errors. Transmit errors include transmit underrun, CTS dropped, CTS time out, and transmit failsafe time out.
`rx_err_thresh`	Specifies the threshold for all receive errors. Receive errors include overrun errors, break/abort errors, framing/cyclic redundancy check (CRC)/frame check sequence (FCS) errors, parity errors, bad frame synchronization, and receive-DMA-buffer-not-allocated errors.
`tx_err_percent`	Specifies the percentage of transmit errors that must occur before a status block is sent to the LLC.
`rx_err_percent`	Specifies the percentage of receive errors that must occur before a status block is sent to the LLC.

Execution Environment

The **CIO_START** operation can be called from the process environment only.

Return Values

EBUSY	Indicates the port state is not opened for a **CIO_START** operation.
EFAULT	Indicates the cross-memory copy service was unsuccessful.
EINVAL	Indicates the physical link parameter is not valid for the port.
EIO	Indicates the device handler could not queue command to the adapter.
ENOMEM	Indicates no mbuf clusters are available.
ENXIO	Indicates the adapter number is out of range.

Related Information

The **ddioctl CIO_GET_FASTWRT** entry point, **mpioctl** entry point.

The **CIO_GET_STAT** mpioctl MPQP Device Handler Operation, **CIO_HALT** mpioctl MPQP Device Handler Operation, **CIO_QUERY** mpioctl MPQP Device Handler Operation, **MP_CHG_PARMS** mpioctl MPQP Device Handler Operation, **MP_START_AR** and **MP_STOP_AR** mpioctl MPQP Device Handler Operations.

MP_CHG_PARMS (Change Parameters) mpioctl MPQP Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Permits the data link control (DLC) to change certain profile parameters after the Multiprotocol Quad Port (MPQP) device has been started.

Description

The **MP_CHG_PARMS** operation permits the DLC to change certain profile parameters after the MPQP device has been started. The *cmd* parameter in the **mpioctl** entry point is set to the **MP_CHG_PARMS** operation. This operation can interfere with communications that are in progress. Data transmission should not be active when this operation is issued.

For this operation, the *extptr* parameter points to a **chng_params** structure. This structure has the following changeable fields:

Field	Description
`chg_mask`	Specifies the mask that indicates which fields are to be changed. The possible choices are: • **CP_POLL_ADDR** • **CP_RCV_TMR** • **CP_SEL_ADDR** More than one field can be changed with one **MP_CHG_PARAMS** operation.
`rcv_timer`	Identifies the timeout value used after transmission of final frames when waiting for receive data in 0.1 second units.
`poll_addr`	Specifies the poll address. Possible values are Synchronous Data Link Control (SDLC) or Binary Synchronous Communications (BSC) poll addresses.
`sel_addr`	Specifies the select address. BSC is the only possible protocol that supports select addresses.

Related Information

The **mpioctl** entry point.

The **CIO_GET_STAT** mpioctl MPQP Device Handler Operation, **CIO_HALT** mpioctl MPQP Device Handler Operation, **CIO_START** mpioctl MPQP Device Handler Operation, **CIO_QUERY** mpioctl MPQP Device Handler Operation, **MP_START_AR** and **MP_STOP_AR** mpioctl MPQP Device Handler Operations.

MP_START_AR (Start Autoresponse) and MP_STOP_AR (Stop Autoresponse) mpioctl MPQP Device Handler Operations

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Permits the Multiprotocol Quad Port (MPQP) to exit or enter Autoresponse mode.

Description

The **MP_START_AR** and **MP_STOP_AR** operations permit the MPQP to enter and exit Autoresponse mode. When the *cmd* parameter is set to the **MP_STOP_AR** operation, the device exits from Autoresponse mode. All received data is sent up to the host. The data link control (DLC) receives an end-of-autoresponse status in the **status_block** structure of the **CIO_GET_STAT** operation.

When the *cmd* parameter is set to the **MP_START_AR** operation, the port is put into Autoresponse mode. The DLC supplies the address and control bytes for receive compare and transmit in the **t_auto_resp** structure pointed to by the *extptr* parameter. This structure contains the following fields:

Field	Description
`rcv_timer`	Identifies the time in 100-msec units that the adapter waits after a frame has been transmitted before reporting an error.
`tx_rx_addr`	Contains the 1-byte address used for compare on the receive frames and as the address byte on transmitted frames.
`tx_cntl`	Specifies the control byte used for transmitted frames.
`rx_cntl`	Identifies the value of control byte on receive frames used for receive compare.

Autoresponse mode is applicable for Synchronous Data Link Control SDLC protocol only. Autoresponse reduces the amount of system overhead during nonproductive link communications. While Data Termination Endpoints (DTEs) are exchange-control information to maintain the link, the adapter can respond to polls from the host without generating any system interrupts.

When in Autoresponse mode, the MPQP adapter compares the receive address and control bytes with the values supplied by the DLC. If a match is found, it generates a response frame with the address and control bytes given in the **MP_START_AR** operation. When a response frame is transmitted, a timer is started with the value given in the `rcv_timer` field. If the adapter does not receive a frame before the timer expires, it detects an error and exits Autoresponse mode.

The following five conditions cause the MPQP adapter to exit Autoresponse mode:

- A receive time out occurs.
- A transmit time out occurs.
- A poll/final frame is received that does not compare with the control data given in the autoresponse operation.
- A fatal link error occurs. Fatal errors include data rate select (DSR) dropped and X.21 cleared received.
- A stop autoresponse command is received from the DLC.

If one of these errors occurs, the adapter exits Autoresponse mode and stays in receive mode. Polls received after these errors occur are passed to the DLC.

The autoresponse operations function with a 4-Port Multiprotocol Interface adapter been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The autoresponse operations can be called from the process environment only.

Return Values

ENOMEM Indicates no mbufs or mbuf clusters are available.
ENXIO Indicates the adapter number is out of range.

Related Information

The **CIO_GET_FASTWRT** ioctl Communications PDH Operation.

The **CIO_GET_STAT** mpioctl MPQP Device Handler Operation, **CIO_HALT** mpioctl MPQP Device Handler Operation, **CIO_QUERY** mpioctl MPQP Device Handler Operation, **CIO_START** mpioctl MPQP Device Handler Operation, **MP_CHG_PARMS** mpioctl MPQP Device Handler Operation.

---

mpmpx Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Allocates and deallocates a channel for the Multiprotocol Quad Port (MPQP) device handler.

Syntax

```
int mpmpx (devno, chanp, channame)
dev_t devno;
int *chanp;
char *channame;
int openflag;
```

Parameters

devno Specifies the major and minor device numbers.

chanp Identifies the channel ID passed as a reference parameter. Unless specified as null, the *channame* parameter is set to the allocated channel ID. If this parameter is null it is set as the ID of the channel to be deallocated.

channame Points to the remaining path name describing the channel to be allocated. There are four possible values:

Equal to NULL
Deallocates the channel.

A pointer to a NULL string
Allows a normal open sequence of the device on the channel ID generated by the **mpmpx** entry point.

D Allows the device to be opened in Diagnostic mode on the channel ID generated by the **mpmpx** entry point.

Pointer to a "W"
Allows the MPQP device to be opened in Diagnostic mode with the adapter in Wrap mode. The device is opened on the channel ID generated by the **mpmpx** entry point.

Description

The **mpmpx** entry point allocates and deallocates a channel. The **mpmpx** entry point is supported similar to the common **ddmpx** entry point.

Return Values

The common return codes for the **mpmpx** entry point are the following:

Return Code	Description
EINVAL	Indicates an invalid parameter.
ENXIO	Indicates the device was open and the Diagnostic mode open request was denied.
EBUSY	Indicates the device was open in Diagnostic mode and the open request was denied.

Related Information

The **ddmpx** entry point, **mpclose** entry point, **mpconfig** entry point, **mpioctl** entry point, **mpopen** entry point, **mpread** entry point, **mpselect** entry point, **mpwrite** entry point.

MPQP Device Handler Interface Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications I/O Subsystem: Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications Physical Device Handler Model Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

mpopen Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Prepares the Multiprotocol Quad Port (MPQP) device for transmitting and receiving data.

Syntax

#include <sys/comio.h>
#include <sys/mpqp.h> int mpopen (*devno***,** *devflag***,** *chan***,** *ext***)**
dev_t *devno***;**
ulong *devflag***;**
int *chan***;**
STRUCT kopen_ext ****ext*;**

Parameters

devno Specifies the major and minor device numbers.

devflag Specifies the flag word. For kernel-mode processes, the *devflag* parameter must be set to the **DKERNEL** flag, which specifies that a kernel routine is making the **mpopen** call. In addition, the following flags can be set:

DWRITE
Specifies to open for reading and writing.

DREAD
Specifies to open for a trace.

DNDELAY
Specifies to open without waiting for the operation to complete. If this flag is set, write requests return immediately and read requests return with 0 length data if no read data is available. The calling process does not sleep. The default is **DELAY** or blocking mode.

DELAY Specifies to wait for the operation to complete before opening. This is the default.

Note: For user-mode processes, the **DKERNEL** flag must be clear.

chan Specifies the channel number assigned by the **mpmpx** entry point.

ext Points to the **kopen_ext** parameter block for kernel-mode processes. Specifies the address to the **mpopen** parameter block for user-mode processes.

Description

The **mpopen** entry point prepares the MPQP device for transmitting and receiving data. This entry point is invoked in response to a **fp_open** kernel service call. The file system in user mode also calls the **mpopen** entry point when an **open** subroutine is issued. The device should be opened for reading and writing data.

Each port on the MPQP adapter must be opened by its own **mpopen** call. Only one open call is allowed for each port. If more than one open call is issued, an error is returned on subsequent **mpopen** calls.

The MPQP device handler only supports one kernel-mode process to open each port on the MPQP adapter. It supports the multiplex (**mpx**) routines and structures compatible with the communications I/O subsystem, but it is not a true multiplexed device.

The kernel process must provide a **kopen_ext** parameter block. This parameter block is found in **/usr/include/sys/comio.h** file.

For a user-mode process, the *ext* parameter points to the **mpopen** structure. This is defined in the **/usr/include/sys/comio.h** file. For calls that do not specify a parameter block, the default values are used.

If adapter features such as the read extended `status` field for binary synchronous communication (BSC) message types as well as other types of information about read data are desired, the *ext* parameter must be supplied. This also requires the **readx** or **read** subroutine. If a system call is used, user data is returned, although status information is not returned. For this reason, it is recommended that **readx** subroutines be used.

The **mpopen** entry point functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Note: A **CIO_START** operation must be issued before the adapter is ready to transmit and receive data. Write commands are not accepted if a **CIO_START** operation has not been completed successfully.

Execution Environment

The **mpopen** entry point can be called from the process environment only.

Return Values

The common return codes for the **mpopen** entry point are the following:

Return Code	Description
ENXIO	Indicates that the port initialization was unsuccessful. This code could also indicate that the registration of the interrupt was unsuccessful.
ECHRNG	Indicates that the channel number is out of range (too high).
ENOMEM	Indicates that there were no mbuf clusters available.
EBUSY	Indicates that the port is in the incorrect state to receive an open call. The port may be already opened or not yet configured.

Related Information

The **mpclose** entry point, **mpconfig** entry point, **mpioctl** entry point, **mpmpx** entry point, **mpread** entry point, **mpselect** entry point, **mpwrite** entry point.

The **read** or **readx** subroutine.

The **fp_open** kernel service.

The **CIO_START** mpioctl MPQP Device Handler Operation.

mpread Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for receiving data from the Multiprotocol Quad Port (MPQP) device.

Syntax

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

int mpread (devno, uiop, chan, ext)
dev_t devno;
struct uio *uiop;
int chan, ext;
```

Parameters

devno	Specifies the major and minor device numbers.
uiop	Pointer to an **uio** structure that provides variables to control the data transfer operation. The **uio** structure is defined in the **/usr/include/sys/uio.h** file.
chan	Specifies the channel number assigned by the **mpmpx** routine.
ext	Specifies the address of the **read_extension** structure. If the *ext* parameter is null, then no parameter block is specified.

Description

Note: Only user-mode processes should use the **mpread** entry point.

The **mpread** entry point provides the means for receiving data from the MPQP device. When a user-mode process user issues a **read** or **readx** subroutine, the kernel calls the **mpread** entry point.

The **DNDELAY** flag, set either at open time or later by an **mpioctl** operation, controls whether **mpread** calls put the caller to sleep pending completion of the call. If a program issues an **mpread** entry point with the **DNDELAY** flag clear (the default), program execution is suspended until the call completes. If the **DNDELAY** flag is set, the call always returns immediately. The user must then issue a poll and a **CIO_GET_STAT** operation to be notified when read data is available.

When user application programs invoke the **mpread** operation through the **read** or **readx** subroutine, the returned length value specifies the number of bytes read. The `status` field in the **read_extension** parameter block should be checked to determine if any errors occurred on the read. One frame is read into each buffer. Therefore, the number of bytes read depends on the size of the frame received.

For a nonkernel process, the device handler copies the data into the buffer specified by the caller. The size of the buffer is limited by the size of the internal buffers on the adapter. If the size of the use buffer exceeds the size of the adapter buffer, the maximum number of bytes on a **mpread** entry point is the size of the internal buffer. For the MPQP adapter, the maximum frame size is defined in the **/usr/include/sys/mpqp.h** file.

Data is not always returned on a read operation when an error occurs. In most cases, the error causes an error log to occur. If no data is returned, the buffer pointer is null. On errors such as buffer overflow, a kernel-mode process receives the error status and the data.

There are also some cases where network data is returned (usually during a **CIO_START** operation). Network data is distinguished from normal receive data by the `status` field in the **read_extension** structure. A nonzero status in this field indicates an error or information about the data.

The MPQP device handler uses a fixed length buffer for transmitting and receiving data. The maximum supported buffer size is 4096 bytes.

The **mpread** entry point functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Note: The MPQP device handler uses fixed length buffers for transmitting and receiving data. The `RX_BUF_LEN` field in the **/usr/include/sys/mpqp.h** file defines the maximum buffer size.

read_extension Parameter Block

For the **mpread** entry points, the *ext* parameter may point to a **read_extension** structure. This structure is found in the **/usr/include/sys/comio.h** file and contains this field:

Field	Description
`status`	Specifies the status of the port. There are six possible values for the returned status parameter. The following status values accompany a data buffer: **CIO_OK** Indicates that the operation was successful. **MP_BUF_OVERFLOW** Indicates receive buffer overflow. For the **MP_BUF_OVERFLOW** value, the data that was received before the buffer overflowed is returned with the overflow status. **MP_X21_CPS** Holds an X.21 call progress signal. **MP_X21_DPI** Holds information provided by X.21 Data Communications Equipment (DCE) (network data). **MP_MODEM_DATA** Contains modem data (for example, an autodial sent by the modem). **MP_AR_DATA_RCVD** Contains data received while in Autoresponse mode.

Note: When using the X.21 physical interface, X.21 centralized multiport (multidrop) operation on a leased-circuit public data network is not supported.

Execution Environment

The **mpread** entry point can be called from the process environment only.

Return Values

The **mpread** entry point returns the number of bytes read. In addition, this entry point may return one of the following:

Return Code	Description
ECHRNG	Indicates the channel number was out of range.
ENXIO	Indicates the port is not in the proper state for a read.
EINTR	Indicates the sleep was interrupted by a signal.
EINVAL	Indicates the read was called by a kernel process.

Related Information

The **mpclose** entry point, **mpconfig** entry point, **mpioctl** entry point, **mpmpx** entry point, **mpopen** entry point, **mpselect** entry point, **mpwrite** entry point.

The **read** or **readx** subroutine.

The **CIO_START** mpioctl operation, **MP_START_AR** mpioctl operation.

The **uio** structure.

Communications Physical Device Handler Model Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

MPQP Device Handler Interface Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications I/O Subsystem: Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

mpselect Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for determining whether specified events have occurred on the Multiprotocol Quad Port (MPQP) device.

Syntax

```
#include <sys/devices.h>
#include <sys/comio.h>

int mpselect (devno, events, reventp, chan)
dev_t devno;
ushort events;
ushort *reventp;
int chan;
```

Parameters

devno	Specifies major and minor device numbers.
events	Identifies the events to check.
reventp	Returns events pointer. This parameter is passed by reference and is used by the **mpselect** entry point to indicate which of the selected events are true at the time of the call.
chan	Specifies the channel number assigned by the **mpmpx** entry point.

Description

Note: Only user-mode processes can use the **mpselect** entry point.

The **mpselect** entry point provides the means for determining if specified events have occurred on the MPQP device. This entry point is supported similar to the **ddselect** communications entry point.

The **mpselect** entry point functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **mpselect** entry point can be called from the process environment only.

Return Values

The common return codes for the **mpselect** entry point are the following:

Return Code	Description
ENXIO	Indicates an attempt to use an unconfigured device.
EINVAL	Indicates the select operation was called from a kernel process.
ECHNG	Indicates the channel number is too large.

Related Information

The **mpclose** entry point, **mpconfig** entry point, **mpioctl** entry point, **mpmpx** entry point, **mpopen** entry point, **mpread** entry point, **mpwrite** entry point.

The **ddselect** communications PDH entry point.

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

MPQP Device Handler Interface Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications I/O Subsystem: Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications Physical Device Handler Model Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

mpwrite Multiprotocol (MPQP) Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for transmitting data to the Multiprotocol Quad Port (MPQP) device.

Syntax

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

int mpwrite (devno, uiop, chan, ext)
dev_t devno;
struct uio *uiop;
int chan, ext;
```

Parameters

devno Specifies major and minor device numbers.

uiop Points to a **uio** structure that provides variables to control the data transfer operation. The **uio** structure is defined in the **/usr/include/sys/uio.h** file.

chan Specifies the channel number assigned by the **mpmpx** entry point.

ext Specifies the address of the **mp_write_extension** parameter block. If the *ext* parameter is null, no parameter block is specified.

Description

The **mpwrite** entry point provides the means for transmitting data to the MPQP device. The kernel calls it when a user-mode process issues a **write** or **writex** subroutine. The **mpwrite** entry point can also be called in response to an **fpwrite** kernel service.

The MPQP device handler uses a fixed length buffer for transmitting and receiving data. The maximum supported buffer size is 4096 bytes.

The **mpwrite** entry point functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

mpwrite Parameter Block

For the **mpwrite** operation, the *ext* parameter points to the **mp_write_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file. The **mp_write_extension** structure contains the following fields:

Field	Description
`status`	Identifies the status of the port. The possible values for the returned `status` field are: **CIO_OK** Indicates the operation was successful. **CIO_TX_FULL** Indicates unable to queue any more transmit requests. **CIO_HARD_FAIL** Indicates hardware failure. **CIO_INV_BFER** Indicates invalid buffer (length equals 0, invalid address). **CIO_NOT_STARTED** Indicates device not yet started.
`write_id`	Contains a user-supplied correlator. The `write_id` field is returned to the caller by the **CIO_GET_STAT** operation if the **CIO_ACK_TX_DONE** flag is selected in the asynchronous status block. For a kernel user, this field is returned to the caller with the **stat_fn** function which was provided at open time.

In addition to the common parameters, the **mp_write_extension** structure contains a field for selecting Transparent mode for binary synchronous communication (BSC). Any nonzero value for this field causes Transparent mode to be selected. Selecting Transparent mode causes the adapter to insert data link escape (DLE) characters before all appropriate control characters. Text sent in Transparent mode is unaltered. Transparent mode is normally used for sending binary files.

Note: If an **mp_write_extension** structure is not supplied, Transparent mode can be implemented by the kernel-mode process by imbedding the appropriate DLE sequences in the data buffer.

Execution Environment

The **mpwrite** entry point can be called from the process environment only.

Return Values

The common return codes for the **mpwrite** entry point are the following:

Return Code	Description
EAGAIN	Indicates that the number of direct memory accesses (DMAs) has reached the maximum allowed or that the device handler cannot get memory for internal control structures. **Note:** The MPQP device handler does not currently support the **tx_fn** function. If a value of **EAGAIN** is returned by an **mpwrite** entry point, the application is responsible for retrying the write.
ECHRNG	Indicates that the channel number is too high.
EINVAL	Indicates one of the following: • The port is not set up properly. • The MPQP device handler could not set up structures for the write. • The port is not valid.
ENOMEM	Indicates that no **mbuf** structure or clusters are available or the total data length is more than a page.
ENXIO	Indicates one of the following: • The port has not been successfully started. • An invalid adapter number was passed. • The specified channel number is illegal.

Related Information

The **mpclose** entry point, **mpconfig** entry point, **mpioctl** entry point, **mpmpx** entry point, **mpopen** entry point, **mpread** entry point, **mpselect** entry point.

The **CIO_GET_STAT** (Get Status) mpioctl MPQP Device Handler Operation.

The **write** or **writex** subroutine.

The **uio** structure.

Communications Physical Device Handler Model Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

MPQP Device Handler Interface Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications I/O Subsystem: Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Binary Synchronous Communication (BSC) with the MPQP Adapter in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

tsclose Multiprotocol (PCI MPQP) Device Handler Entry Point

Purpose

Resets the IBM ARTIC960Hx adapter (PCI MPQP) and device handler to a known state and returns system resources back to the system on the last close for that adapter.

Syntax

```
int tsclose (devno, chan, ext)
dev_t devno;
int chan, ext;
```

Description

The **tsclose** entry point routine resets the PCI MPQP adapter to a known state and returns system resources to the system on the last close for that adapter. The port no longer accepts **tsread**, **tswrite**, or **tsioctl** operation requests. The **tsclose** entry point is called in user mode by issuing a **close** system call. The **tsclose** entry point is invoked in response to an **fp_close** kernel service.

On an **tsclose** entry point, the PCI MPQP device handler does the following:

- Frees all internal data areas for the corresponding **tsopen** entry point.
- Purges receive data queued for this **tsopen** entry point.

On the last **tsclose** entry point for a particular adapter, the PCI MPQP device handler also does the following:

- Frees its interrupt level to the system.
- Frees the DMA channel.
- Disables adapter interrupts.
- Sets all internal data elements to their default settings.

The **tsclose** entry point closes the device. For each **tsopen** entry point issued, there must be a corresponding **tsclose** entry point.

Before issuing the **tsclose** entry point, the caller should issue a **CIO_HALT** operation for each **CIO_START** operation issued during that particular instance of open. If a close request is received without a preceding **CIO_HALT** operation, the functions of the halt are performed. A close request without a preceding **CIO_HALT** operation occurs only during abnormal termination of the port.

The **tsclose** entry point functions with a 4-port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Parameters

devno	Specifies major and minor device numbers.
chan	Specifies the channel number assigned by the **tsmpx** entry point.
ext	Ignored by the PCI MPQP device handler.

Execution Environment

The **tsclose** entry point can be called from the process environment only.

Return Values

The common return codes for the **tsclose** entry point are:

ECHRNG	Indicates the channel number is too large.
ENXIO	Indicates the port initialization was unsuccessful. This code could also indicate that the registration of the interrupt was unsuccessful.
ECHRNG	Indicates the channel number is out of range (too high).

Related Information

The**tsconfig** entry point, **tsioctl** entry point, **tsmpx** entry point, **tsopen** entry point, **tsread** entry point, **tsselect** entry point, **tswrite** entry point.

The **CIO_HALT** tsioctl PCI MPQP Device Handler Operation, **CIO_START** tsioctl PCI MPQP Device Handler Operation.

The **close** system call.

The **fp_close** kernel service.

---

tsconfig Multiprotocol (PCI MPQP) Device Handler Entry Point

Purpose

Provides functions for initializing and terminating the IBM ARTIC960Hx PCI adapter (PCI MPQP) and device handler.

Syntax

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

int tsconfig (devno, cmd, uiop)
dev_t devno;
int cmd;
struct uio *uiop;
```

Description

The **tsconfig** entry point provides functions for initializing and terminating the PCI MPQP device handler and adapter. It is invoked through the **/usr/include/sys/config** device driver at device configuration time. This entry point supports the following operations:

- **CFG_INIT**
- **CFG_TERM**

The **tsconfig** entry point functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Parameters

devno Specifies major and minor device numbers.

cmd Specifies the function to be performed by this routine. There are two possible functions:

CFG_INIT
Initializes device handler and internal data areas.

CFG_TERM
Terminates the device handler.

uiop Points to a **uio** structure. The **uio** structure is defined in the **/usr/include/sys/uio.h** file.

Execution Environment

The **tsconfig** entry point can be called from the process environment only.

Related Information

The **tsclose** entry point, **tsioctl** entry point, **tsmpx** entry point, **tsopen** entry point, **tsread** entry point, **tsselect** entry point, **tswrite** entry point.

The **ddconfig** routine.

PCI MPQP Device Handler Interface Overview.

Communications I/O Subsystem: Programming Introduction.

---

tsioctl Multiprotocol (PCI MPQP) Device Handler Entry Point

Purpose

Provides various functions for controlling the IBM ARTIC960Hx PCI adapter (PCI MPQP) and device handler.

Syntax

```
#include <sys/devinfo.h>
#include <sys/ioctl.h>
#include <sys/comio.h>
#include <sys/mpqp.h>

int tsioctl
(devno, cmd, extptr, devflag, chan, ext)
dev_t devno;
int cmd, extptr;
ulong devflag;
int chan, ext;
```

Description

The **tsioctl** entry point provides various functions for controlling the PCI MPQP adapter. There are 16 valid **tsioctl** operations, including:

CIO_GET_STAT	Gets the status of the current PCI MPQP adapter and device handler.
CIO_HALT	Ends a session with the PCI MPQP device handler.
CIO_START	Initiates a session with the PCI MPQP device handler.
CIO_QUERY	Reads the counter values accumulated by the PCI MPQP device handler.
MP_CHG_PARMS	Permits the DLC to change certain profile parameters after the PCI MPQP device has been started.

The **tsioctl** entry point functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

The possible **tsioctl** operation codes can be found in the **/usr/include/sys/ioctl.h**, **/usr/include/sys/comio.h**, and **/usr/include/sys/mpqp.h** files.

Parameters

devno	Specifies major and minor device numbers.
cmd	Identifies the operation to be performed.
extptr	Specifies an address of the parameter block.
devflag	Allows **tsioctl** calls to inherit properties that were specified at open time. The PCI MPQP device handler inspects the **DNDELAY** flag for ioctl calls. Kernel-mode data link control (DLC) sets the **DKERNEL** flag that must be set for a **tsopen** call.
chan	Specifies the channel number assigned by the **tsmpx** entry point.
ext	Not used by PCI MPQP device handler.

Execution Environment

The **tsioctl** entry point can be called from the process environment only.

Return Values

The common return codes for the **tsioctl** entry point are:

ENOMEM Indicates the no memory buffers (mbufs) or mbuf clusters are available.
ENXIO Indicates the adapter number is out of range.

Related Information

The **tsclose** entry point, **tsconfig** entry point, **tsmpx** entry point,, **tsopen** entry point, **tsread** entry point, **tsselect** entry point, **tswrite** entry point.

The **CIO_GET_STAT** tsioctl PCI MPQP Device Handler Operation, **CIO_HALT** tsioctl PCI MPQP Device Handler Operation, **CIO_QUERY** tsioctl PCI MPQP Device Handler Operation, **CIO_START** tsioctl PCI MPQP Device Handler Operation, **MP_CHG_PARMS** tsioctl PCI MPQP Device Handler Operation.

CIO_GET_STAT (Get Status) tsioctl PCI MPQP Device Handler Operation

Purpose

Gets the status of the current IBM ARTIC960Hx PCI adapter (PCI MPQP) and device handler.

Description

Note: Only user-mode processes can use the **CIO_GET_STAT** operation.

The **CIO_GET_STAT** operation gets the status of the current PCI MPQP adapter and device handler. For the PCI MPQP device handler, both solicited and unsolicited status can be returned.

Solicited status is status information that is returned as a completion status to a particular operation. The **CIO_START**, **CIO_HALT**, and **tswrite** operations all have solicited status returned. However, for many asynchronous events common to wide-area networks, these are considered unsolicited status. The asynchronous events are divided into three classes:

- Hard failures
- Soft failures
- Informational (or status-related) messages

The **CIO_GET_STAT** operation functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Status Blocks for the Multiprotocol Device Handler

For the **CIO_GET_STAT** operation, the *extptr* parameter points to a **status_block** structure. When returned, the device handler fills this structure with the appropriate information. The **status_block** structure is defined in the **/usr/include/sys/comio.h** file and returns one of the possible status conditions:

Status blocks are used to communicate status and exception information to user-mode processes.

User-mode processes receive a status block whenever they request a **CIO_GET_STAT** operation. A user-mode process can wait for the next available status block by issuing a **tsselect** entry point with the specified **POLLPRI** event.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, **CIO_START_DONE**). The following possible PCI MPQP status blocks exist:

- **CIO_ASYNC_STATUS**
- **CIO_HALT_DONE**
- **CIO_START_DONE**
- **CIO_TX_DONE**
- **MP_THRESH_EXC**

CIO_ASYNC_STATUS Status Block

Asynchronous status notifies the data link control of asynchronous events such as network and adapter failures.

Code	CIO_ASYNC_STATUS
option[0]	Can be one of the following:
	MP_DSR_DROPPED, MP_RCV_TIMEOUT, MP_RELOAD_CMPL, MP_RESET_CMPL, MP_X21_CLEAR
option[1]	Not used
option[2]	Not used
option[3]	Not used

Note: The **MP_RELOAD_C** and **MPLMP_RESET_CMPL** values are for diagnostic use only.

CIO_HALT_DONE Status Block

The **CIO_HALT** operation ends a session with the PCI MPQP device handler. On a successfully completed Halt Device operation, the following status block is provided:

Code	CIO_HALT_DONE
option[0]	**CIO_OK**
option[1]	**MP_FORCED_HALT or MP_NORMAL_HALT**
option[2]	**MP_NETWORK_FAILURE or MP_HW_FAILURE**

A *forced halt* is a halt completed successfully in terms of the data link control is concerned, but terminates forcefully because of either an adapter error or a network error. This is significant for X.21 or other switched networks where customers can be charged if the call does not disconnect properly.

Note: When using the X.21 physical interface, X.21 centralized multiport (multidrop) operation on a leased-circuit public data network is not supported.

CIO_START_DONE Status Block

On a successfully completed **CIO_START** operation, the following status block is provided:

Code	CIO_START_DONE
option[0]	**CIO_OK**
option[1]	Network ID
option[2]	Not used
option[3]	Not used

On an unsuccessful Start Device **CIO_START** tsioctl operation, the following status block is provided:

Code	CIO_START_DONE
option[0]	Can be one of the following: **MP_ADAP_NOT_FUNC** Adapter not functional **MP_TX_FAILSAFE_TIMEOUT** Transmit command did not complete. **MP_DSR_ON_TIMEOUT** DSR failed to come on. **MP_DSR_ALRDY_ON** DSR already on for a switched line. **MP_X21_CLEAR** Unexpected clear received from the DCE.
option[1]	If the option[0] field is set to **MP_X21_TIMEOUT**, the option[1] field contains the specific X.21 timer that expired.
option[2]	Not used.
option[3]	Not used.

CIO_TX_DONE Status Block

On completion of a multiprotocol transmit, the following status block is provided:

Code	CIO_TX_DONE
option[0]	Can be one of the following: **CIO_OK** **MP_TX_UNDERRUN** **MP_X21_CLEAR** **MP_TX_FAILSAFE_TIMEOUT** The transmit command did not complete. **MP_TX_ABORT** Transmit aborted due to CIO_HALT operation.
option[1]	Identifies the write_id field supplied by the caller in the write command if **TX_ACK** was selected.
option[2]	Points to the buffer with transmit data.
option[3]	Not used.

MP_THRESH_EXC Status Block

A threshold for one of the counters defined in the start profile has reached its threshold.

Code	MP_THRESH_EXC
option[0]	Indicates the expired threshold.
	The following values are returned to indicate the threshold that was exceeded: **MP_TOTAL_TX_ERR, MP_TOTAL_RX_ERR, MP_TX_PERCENT, MP_RX_PERCENT**
option[1]	Not used.
option[2]	Not used.

Code	MP_THRESH_EXC
option[3]	Not used.

Execution Environment

The **CIO_GET_STAT** operation can be called from the process environment only.

Return Values

The return codes for the **CIO_GET_STAT** operation are:

ENOMEM Indicates no mbufs or mbuf clusters are available.
ENXIO Indicates the adapter number is out of range.

Related Information

The **CIO_HALT** tsioctl PCI MPQP Device Handler Operation, **CIO_QUERY** tsioctl PCI MPQP Device Handler Operation, **CIO_START** tsioctl PCI MPQP Device Handler Operation, **MP_CHG_PARMS** tsioctl PCI MPQP Device Handler Operation.

The **tsioctl** entry point, **tswrite** entry point.

CIO_HALT (Halt Device) tsioctl PCI MPQP Device Handler Operation

Purpose

Ends a session with the IBM ARTIC960Hx PCI adapter (PCI MPQP) and device handler and terminates the connection to the PCI MPQP link.

Description

The **CIO_HALT** operation terminates a session with the PCI MPQP device handler. The caller specifies which network ID to halt. The **CIO_HALT** operation removes the network ID from the network ID table and disconnects the physical link. A **CIO_HALT** operation must be issued for each **CIO_START** operation that completed successfully.

Data received for the specified network ID before the **CIO_HALT** operation is called can be retrieved by the caller using the **tsselect** and **tsread** entry points.

If the **CIO_HALT** operation terminates abnormally, the status is returned either asynchronously or as part of the **CIO_HALT_DONE**. Whatever the case, the **CIO_GET_STAT** operation is used to get information about the error. When a halt is terminated abnormally (for example, due to network failure), the following occurs:

- The link is terminated.
- The drivers and receivers are disabled for the indicated port.
- The port can no longer transmit or receive data.

No recovery procedure is required by the caller; however, logging the error is required.

Errors are reported on halt operations because the user could continue to be charged for connect time if the network does not recognize the halt. This error status permits a network application to be notified about an abnormal link disconnection and then take corrective action, if necessary.

The **CIO_HALT** operation functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Parameter Block

For the PCI MPQP **CIO_HALT** operation, the *extptr* parameter points to a **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

`status` Specifies the status of the port. This field is set for immediately detectable errors. Possible values for the status filed are:

- **CIO_OK**
- **CIO_NETID_INV**

If the calling process does not wish to sleep while the halt is in progress, the **DNDELAY** option can be used. In either case, the status of the halt is retrieved using the **CIO_GET_STAT** operation and a **CIO_HALT_DONE** status block is returned. The **CIO_HALT_DONE** status block should be used as an indication of completion.

`netid` Contains the network ID the caller wishes to halt. The network ID is placed in the least significant byte of the `netid` field.

Execution Environment

The **CIO_HALT** operation can be called from the process environment only.

Return Values

The **CIO_HALT** operation returns common communications return values. In addition, the following PCI MPQP specific errors may be returned:

EBUSY Indicates the device is not started or is not in a data transfer state.
ENOMEM Indicates there are no mbufs or mbuf clusters available.
ENXIO Indicates the adapter number is out of range.

Files

/usr/include/sys/comio.h Contains the **session_blk** structure definition.

Related Information

The **tsread** entry point, **tsselect** entry point.

The **CIO_GET_STAT** tsioctl PCI MPQP Device Handler Operation, **CIO_QUERY** tsioctl PCI MPQP Device Handler Operation, **CIO_START** tsioctl PCI MPQP Device Handler Operation, **MP_CHG_PARMS** tsioctl PCI MPQP Device Handler Operation.

Status Blocks for the Multiprotocol Device Handler.

CIO_QUERY (Query Statistics) tsioctl PCI MPQP Device Handler Operation

Purpose

Provides the means to read counter values accumulated by the IBM ARTIC960Hx PCI adapter (PCI MPQP) and device handler.

Description

The **CIO_QUERY** operation reads the counter values accumulated by the PCI MPQP device handler. The counters are initialized to 0 by the first **tsopen** entry point operation.

The **CIO_QUERY** operation returns the Reliability/Availability/Serviceability field of the define device structure (DDS).

The **CIO_QUERY** operation functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

The t_query_parms Parameter Block

For this operation, the *extptr* parameter points to an **t_query_parms** structure. This structure is defined in the **/usr/include/sys/mpqp.h** file and has the following fields:

`status`	Contains additional information about the completion of the status block. Device-dependent codes may also be returned.
`CIO_OK`	Indicates that the operation was successful.
`bufptr`	Specifies the address of a buffer where the returned statistics are to be placed.
`buflen`	Specifies the length of the buffer; it should be at least 45 words long (unsigned long).
`reserve`	Reserved for use in future releases.

Statistics Logged for PCI MPQP Ports

The following statistics are logged for each PCI MPQP port.

- Bytes transmitted
- Bytes received
- Frames transmitted
- Frames received
- Receive errors
- Transmission errors
- DMA buffer not large enough or not allocated
- CTS time out
- CTS dropped during transmit
- DSR time out
- DSR dropped
- DSR on before DTR on a switched line
- DCE clear during call establishment
- DCE clear during data phase
- X.21 T1-T5 time outs
- X.21 invalid DCE-provided information (DPI)

Note: When using the X.21 physical interface, X.21 centralized multiport (multidrop) operation on a leased-circuit public data network is not supported.

Execution Environment

The **CIO_QUERY** operation can be called from the process environment only.

Return Values

EFAULT	Indicates a specified address is not valid.
EINVAL	Indicates a parameter is not valid.

EIO Indicates an error has occurred.
ENOMEM Indicates the operation was unable to allocate the required memory.
ENXIO Indicates an attempt to use unconfigured device.

Related Information

The **tsioctl** entry point, **tsopen** entry point.

The **CIO_GET_STAT** tsioctl PCI MPQP Device Handler Operation, **CIO_HALT** tsioctl PCI MPQP Device Handler Operation, **CIO_START** tsioctl PCI MPQP Device Handler Operation, **MP_CHG_PARMS** tsioctl PCI MPQP Device Handler Operation.

CIO_START (Start Device) tsioctl PCI MPQP Device Handler Operation

Purpose

Starts a session with the IBM ARTIC960Hx PCI (PCI MPQP) device handler.

Description

The **CIO_START** operation registers a network ID in the network ID table and establishes the physical connection with the PCI MPQP device. Once this start operation completes successfully, the port is ready to transmit and receive data.

Note: The **CIO_START** operation defines the protocol- and configuration-specific attributes of the selected port. All bits that are not defined must be set to 0 (zero).

For the PCI MPQP **CIO_START** operation, the *extptr* parameter points to a **t_start_dev** structure. This structure contains pointers to the **session_blk** structure.

The **session_blk** structure contains the `netid` and `status` fields. The **t_start_dev** device-dependent information for an PCI MPQP device follows the session block. All of these structures can be found in the **/usr/include/sys/mpqp.h** file.

The **CIO_START** operation functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

t_start_dev Fields

The **t_start_dev** structure contains the following fields:

`phys_link` Indicates the physical link protocol. Only one type of physical link is valid at a time. The supported values are:

Physical Link Type	
PL_232D	EIA-232D
PL_V35	V.35
PL_X21	X.21

Note: When using the X.21 physical interface, X.21 centralized multiport (multidrop) operation on a leased-circuit public data network is not supported.

`dial_proto` The `dial_proto` field is ignored.

`data_proto` Identifies the possible data protocol selections during the data transfer phase. The `data_flags` field has different meanings depending on what protocol is selected. The `data_proto` field accepts the following values:

DATA_PRO_BSC
Indicates a bisync protocol.

DATA_PRO_SDLC_FDX
Indicates receivers enabled during transmit.

DATA_PRO_SDLC_HDX
Indicates receivers disabled during transmit.

`modem_flags` Establishes modem characteristics. This field accepts the following values:

MF_AUTO
Indicates that the call is to be answered or dialed automatically.

MF_CALL
Indicates an outgoing call.

MF_LEASED
Indicates a leased telephone circuit.

MF_LISTEN
Indicates an incoming call (switched only).

MF_MANUAL
Indicates that the operator answers or dials the call manually.

MF_SWITCHED
Indicates a switched telephone circuit.

Note: Since each of these modem chracteristics are handled by the modem, the driver actually determines connection status in the same way, no matter what value is set in the *modem_flags* field. When the **CIO_START** ioctl is executed, the DTR signal is asserted and an active connection is reported when an active DSR signal is detected.

`poll_addr` Identifies the address-compare value for a Binary Synchronous Communication (BSC) polling frame or an Synchronous Data Link Control (SDLC) frame. If using BSC, a value for the selection address must also be provided or the address-compare is not enabled. If a frame is received that does not match the poll address (or select address for BSC), the frame is not passed to the system.

`select_addr` Specifies a valid select address for BSC only.

`modem_int_mask`	Reserved. This value must be 0.
`baud_rate`	This value should be set to 0 to indicate the port is to be externally clocked (that is, use modem clocking).
`rcv_timeout`	Indicates the period of time, expressed in 100-msec units (0.10 sec), used for setting the receive timer. The PCI MPQP device driver starts the receive timer whenever the **CIO_START** operation completes and a final transmit occurs. If a receive occurs that is not a receive final frame, the timer is restarted. The timer is stopped when the receive final occurs. If the timer expires before a receive occurs, an error is reported to the logical link control (LLC) protocol. After the **CIO_START** operation completes, the receive time out value can be changed by the **MP_CHG_PARMS** operation. A value of zero indicates that a receive timer should not be activated. Final frames in SDLC are all frames with the poll or final bit set. In BSC, all frames are final frames except intermediate text block (ITB) frames.
`rcv_data_offset`	Reserved
`dial_data_length`	Not used.

Flag Fields for Protocols

Flag fields in the **t_start_dev** structure take different values depending on the type of protocol selected.

Data Flags for the BSC Protocol

If BSC is selected in the `data_proto` field, either ASCII or EBCDIC character sets can be used. Control characters are stripped automatically on reception. Data link escape (DLE) characters are automatically inserted and deleted in transparent mode. If BSC Address Check mode is selected, values for both poll and select addresses must be supplied. Odd parity is used if ASCII is selected.

The following are the default values:

- EBCDIC.
- Do not restart the receive timer.
- Do not check addresses.
- RTS controlled.

The data flags for the BSC protocol are:

DATA_FLG_ADDR_CHK	Address-compare select. This causes frames to be filtered by the hardware based on address. Only frames with matching addresses are sent to the system.
DATA_FLG_BSC_ASC	ASCII BSC select.
DATA_FLG_C_CARR_ON	Continuous carrier (RTS always on).
DATA_FLG_C_CARR_OFF	RTS-disabled between transmits (default).

Data Flags for the SDLC Protocol

For the Synchronous Data Link Control (SDLC) protocol, the flag for NRZ or NRZI must match the data-encoding method that is used by the remote DTE. If SDLC Address Check mode is selected, the poll address byte must also be specified. The receive timer (RT) is started whenever a final block is transmitted. If RT is set to 1, the receive timer is restarted after expiration. If RT is set to 0, the receive timer is not restarted after expiration. The receive timer value is specified by the 16-bit `rcv_timeout` field. The following are the acceptable SDLC data flags:

DATA_FLG_NRZI	NRZI select (default is NRZ).
DATA_FLG_ADDR_CHK	Address-compare select.
DATA_FLG_RST_TMR	Restart receive timer.

DATA_FLG_C_CARR_ON	Continuous carrier (RTS always on).
DATA_FLG_C_CARR_OFF	RTS disabled between transmits (default).

t_err_threshold Fields

The **t_err_threshold** structure describes the format for defining thresholds for transmit and receive errors. Counters track the total number of transmit and receive errors. Individual counters track certain types of errors. Thresholds can be set for individual errors, total errors, or a percentage of transmit and receive errors from all frames received.

When a counter reaches its threshold value, a status block is returned by the driver. The status block indicates the type of error counter that reached its threshold. If multiple thresholds are reached at the same time, the first expired threshold in the list is reported as having expired and its counter is reset to 0. The user can issue a **CIO_QUERY** operation call to retrieve the values of all counters.

If no thresholding is desired, the threshold should be set to 0. A value of 0 indicates that LLC should not be notified of an error at any time. To indicate that the LLC should be notified of every occurrence of an error, the threshold should be set to 1.

The **t_err_threshold** structure contains the following fields:

`tx_err_thresh`	Specifies the threshold for all transmit errors. Transmit errors include transmit underrun, CTS dropped, CTS time out, and transmit failsafe time out.
`rx_err_thresh`	Specifies the threshold for all receive errors. Receive errors include overrun errors, break/abort errors, framing/cyclic redundancy check (CRC)/frame check sequence (FCS) errors, parity errors, bad frame synchronization, and receive-DMA-buffer-not-allocated errors.
`tx_err_percent`	Specifies the percentage of transmit errors that must occur before a status block is sent to the LLC.
`rx_err_percent`	Specifies the percentage of receive errors that must occur before a status block is sent to the LLC.

Execution Environment

The **CIO_START** operation can be called from the process environment only.

Return Values

CIO_OK	Indicates successful **CIO_START** operation.
EBUSY	Indicates the port state is not opened for a **CIO_START** operation.
EFAULT	Indicates the cross-memory copy service was unsuccessful.
EINVAL	Indicates the physical link parameter is not valid for the port.
EIO	Indicates the device handler could not queue command to the adapter.
ENOMEM	Indicates no mbuf clusters are available.
ENXIO	Indicates the adapter number is out of range.

Related Information

The **tsioctl** entry point.

The **CIO_GET_STAT** tsioctl PCI MPQP Device Handler Operation, **CIO_HALT** tsioctl PCI MPQP Device Handler Operation, **CIO_QUERY** tsioctl PCI MPQP Device Handler Operation, **MP_CHG_PARMS** tsioctl PCI MPQP Device Handler Operation.

MP_CHG_PARMS (Change Parameters) tsioctl PCI MPQP Device Handler Operation

Purpose

Permits the data link control (DLC) to change certain profile parameters after the IBM ARTIC960Hx PCI (PCI MPQP) device has been started.

Description

The **MP_CHG_PARMS** operation permits the DLC to change certain profile parameters after the PCI MPQP device has been started. The *cmd* parameter in the **tsioctl** entry point is set to the **MP_CHG_PARMS** operation. This operation can interfere with communications that are in progress. Data transmission should not be active when this operation is issued.

For this operation, the *extptr* parameter points to a **t_chg_parms** structure. This structure has the following changeable fields:

`chg_mask` Specifies the mask that indicates which fields are to be changed. The possible choices are:
- **CP_POLL_ADDR**
- **CP_RCV_TMR**
- **CP_SEL_ADDR**

More than one field can be changed with one **MP_CHG_PARMS** operation.

`rcv_timer` Identifies the timeout value used after transmission of final frames when waiting for receive data in 0.1 second units.

`poll_addr` Specifies the poll address. Possible values are Synchronous Data Link Control (SDLC) or Binary Synchronous Communications (BSC) poll addresses.

`select_addr` Specifies the select address. BSC is the only possible protocol that supports select addresses.

Related Information

The **tsioctl** entry point.

The **CIO_GET_STAT** tsioctl PCI MPQP Device Handler Operation, **CIO_HALT** tsioctl PCI MPQP Device Handler Operation, **CIO_START** tsioctl PCI MPQP Device Handler Operation, **CIO_QUERY** tsioctl PCI MPQP Device Handler Operation.

tsmpx Multiprotocol (PCI MPQP) Device Handler Entry Point

Purpose

Allocates and deallocates a channel for the IBM ARTIC960Hx PCI (PCI MPQP) device handler.

Syntax

```
int tsmpx (devno, chanp, channame)
dev_t devno;
int *chanp;
char *channame;
int openflag;
```

Description

The **tsmpx** entry point allocates and deallocates a channel. The **tsmpx** entry point is supported similar to the common **ddmpx** entry point.

Parameters

devno Specifies the major and minor device numbers.

chanp Identifies the channel ID passed as a reference parameter. Unless specified as null, the *channame* parameter is set to the allocated channel ID. If this parameter is null it is set as the ID of the channel to be deallocated.

channame Points to the remaining path name describing the channel to be allocated. There are four possible values:

Equal to NULL
Deallocates the channel.

A pointer to a NULL string
Allows a normal open sequence of the device on the channel ID generated by the **tsmpx** entry point.

Return Values

The common return codes for the **tsmpx** entry point are the following:

EINVAL Indicates an invalid parameter.
ENXIO Indicates the device was open and the Diagnostic mode open request was denied.
EBUSY Indicates the device was open in Diagnostic mode and the open request was denied.

Related Information

The **ddmpx** entry point, **tsclose** entry point, **tsconfig** entry point, **tsioctl** entry point, **tsopen** entry point, **tsread** entry point, **tsselect** entry point, **tswrite** entry point.

PCI MPQP Device Handler Interface Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications I/O Subsystem: Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications Physical Device Handler Model Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

tsopen Multiprotocol (PCI MPQP) Device Handler Entry Point

Purpose

Prepares the IBM ARTIC960Hx PCI (PCI MPQP) device for transmitting and receiving data.

Syntax

```
#include <sys/comio.h>
#include <sys/mpqp.h>

int tsopen  (devno, devflag, chan, ext)
dev_t devno;
ulong devflag;
int chan;
STRUCT kopen_ext *ext;
```

Description

The **tsopen** entry point prepares the PCI MPQP device for transmitting and receiving data. This entry point is invoked in response to a **fp_open** kernel service call. The file system in user mode also calls the **tsopen** entry point when an **open** subroutine is issued. The device should be opened for reading and writing data.

Each port on the PCI MPQP adapter must be opened by its own **tsopen** call. Only one open call is allowed for each port. If more than one open call is issued, an error is returned on subsequent **tsopen** calls.

The PCI MPQP device handler only supports one kernel-mode process to open each port on the PCI MPQP adapter. It supports the multiplex (**mpx**) routines and structures compatible with the communications I/O subsystem, but it is not a true multiplexed device.

The kernel process must provide a **kopen_ext** parameter block. This parameter block is found in **/usr/include/sys/comio.h** file.

For a user-mode process, the *ext* parameter points to the **tsopen** structure. This is defined in the **/usr/include/sys/comio.h** file. For calls that do not specify a parameter block, the default values are used.

If adapter features such as the read extended `status` field for binary synchronous communication (BSC) message types as well as other types of information about read data are desired, the *ext* parameter must be supplied. This also requires the **readx** or **read** subroutine. If a system call is used, user data is returned, although status information is not returned. For this reason, it is recommended that **readx** subroutines be used.

The **tsopen** entry point functions with a 4-Port Multiprotocol Interface Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Note: A **CIO_START** operation must be issued before the adapter is ready to transmit and receive data. Write commands are not accepted if a **CIO_START** operation has not been completed successfully.

Parameters

devno Specifies the major and minor device numbers.

devflag Specifies the flag word. For kernel-mode processes, the *devflag* parameter must be set to the **DKERNEL** flag, which specifies that a kernel routine is making the **tsopen** call. In addition, the following flags can be set:

DWRITE
Specifies to open for reading and writing.

DREAD
Specifies to open for a trace.

DNDELAY
Specifies to open without waiting for the operation to complete. If this flag is set, write requests return immediately and read requests return with 0 length data if no read data is available. The calling process does not sleep. The default is **DELAY** or blocking mode.

DELAY Specifies to wait for the operation to complete before opening. This is the default.

Note: For user-mode processes, the **DKERNEL** flag must be clear.

chan Specifies the channel number assigned by the **tsmpx** entry point.

ext Points to the **kopen_ext** parameter block for kernel-mode processes. Specifies the address to the **tsopen** parameter block for user-mode processes.

Execution Environment

The **tsopen** entry point can be called from the process environment only.

Return Values

The common return codes for the **tsopen** entry point are the following:

ENXIO	Indicates that the port initialization was unsuccessful. This code could also indicate that the registration of the interrupt was unsuccessful.
ECHRNG	Indicates that the channel number is out of range (too high).
ENOMEM	Indicates that there were no mbuf clusters available.
EBUSY	Indicates that the port is in the incorrect state to receive an open call. The port may be already opened or not yet configured.

Related Information

The **tsclose** entry point, **tsconfig** entry point, **tsioctl** entry point, **tsmpx** entry point, **tsread** entry point, **tsselect** entry point, **tswrite** entry point.

The **read** or **readx** subroutine.

The **fp_open** kernel service.

The **CIO_START** tsioctl PCI MPQP Device Handler Operation.

---

tsread Multiprotocol (PCI MPQP) Device Handler Entry Point

Purpose

Provides the means for receiving data from the IBM ARTIC960Hx PCI (PCI MPQP) device.

Syntax

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

int tsread (devno, uiop, chan, ext)
dev_t devno;
struct uio *uiop;
int chan, ext;
```

Description

Note: Only user-mode processes should use the **tsread** entry point.

The **tsread** entry point provides the means for receiving data from the PCI MPQP device. When a user-mode process user issues a **read** or **readx** subroutine, the kernel calls the **tsread** entry point.

The **DNDELAY** flag, set either at open time or later by an **tsioctl** operation, controls whether **tsread** calls put the caller to sleep pending completion of the call. If a program issues an **tsread** entry point with the **DNDELAY** flag clear (the default), program execution is suspended until the call completes. If the **DNDELAY** flag is set, the call always returns immediately. The user must then issue a poll and a **CIO_GET_STAT** operation to be notified when read data is available.

When user application programs invoke the **tsread** operation through the **read** or **readx** subroutine, the returned length value specifies the number of bytes read. The `status` field in the **read_extension** parameter block should be checked to determine if any errors occurred on the read. One frame is read into each buffer. Therefore, the number of bytes read depends on the size of the frame received.

For a nonkernel process, the device handler copies the data into the buffer specified by the caller. The size of the buffer is limited by the size of the internal buffers on the adapter. If the size of the use buffer exceeds the size of the adapter buffer, the maximum number of bytes on a **tsread** entry point is the size of the internal buffer. For the PCI MPQP adapter, the maximum frame size is defined in the **/usr/include/sys/mpqp.h** file.

Data is not always returned on a read operation when an error occurs. In most cases, the error causes an error log to occur. If no data is returned, the buffer pointer is null. On errors such as buffer overflow, a kernel-mode process receives the error status and the data.

There are also some cases where network data is returned (usually during a **CIO_START** operation). Network data is distinguished from normal receive data by the `status` field in the **read_extension** structure. A nonzero status in this field indicates an error or information about the data.

The PCI MPQP device handler uses a fixed length buffer for transmitting and receiving data. The maximum supported buffer size is 4096 bytes.

The **tsread** entry point functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Note: The PCI MPQP device handler uses fixed length buffers for transmitting and receiving data. The `RX_BUF_LEN` field in the **/usr/include/sys/mpqp.h** file defines the maximum buffer size.

read_extension Parameter Block

For the **tsread** entry points, the *ext* parameter may point to a **read_extension** structure. This structure is found in the **/usr/include/sys/comio.h** file and contains this field:

`status` Specifies the status of the port. There are six possible values for the returned status parameter. The following status values accompany a data buffer:

CIO_OK
Indicates that the operation was successful.

MP_BUF_OVERFLOW
Indicates receive buffer overflow. For the **MP_BUF_OVERFLOW** value, the data that was received before the buffer overflowed is returned with the overflow status.

Note: When using the X.21 physical interface, X.21 centralized multiport (multidrop) operation on a leased-circuit public data network is not supported.

Parameters

devno Specifies the major and minor device numbers.

uiop Pointer to an **uio** structure that provides variables to control the data transfer operation. The **uio** structure is defined in the **/usr/include/sys/uio.h** file.

chan Specifies the channel number assigned by the **tsmpx** routine.

ext Specifies the address of the **read_extension** structure. If the *ext* parameter is null, then no parameter block is specified.

Execution Environment

The **tsread** entry point can be called from the process environment only.

Return Values

The **tsread** entry point returns the number of bytes read. In addition, this entry point may return one of the following:

ECHRNG	Indicates the channel number was out of range.
ENXIO	Indicates the port is not in the proper state for a read.
EINTR	Indicates the sleep was interrupted by a signal.
EINVAL	Indicates the read was called by a kernel process.

Related Information

The **tsclose** entry point, **tsconfig** entry point, **tsioctl** entry point, **tsmpx** entry point, **tsopen** entry point, **tsselect** entry point, **tswrite** entry point.

The **read** or **readx** subroutine.

The **CIO_START** tsioctl operation.

The **uio** structure.

Communications Physical Device Handler Model Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

PCI MPQP Device Handler Interface Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications I/O Subsystem: Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

tsselect Multiprotocol (PCI MPQP) Device Handler Entry Point

Purpose

Provides the means for determining whether specified events have occurred on the IBM ARTIC960Hx PCI (PCI MPQP) device.

Syntax

```
#include <sys/devices.h>
#include <sys/comio.h>

int tsselect (devno, events, reventp, chan)
dev_t devno;
ushort events;
ushort *reventp;
int chan;
```

Description

Note: Only user-mode processes can use the **tsselect** entry point.

The **tsselect** entry point provides the means for determining if specified events have occurred on the PCI MPQP device. This entry point is supported similar to the **ddselect** communications entry point.

The **tsselect** entry point functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Parameters

devno	Specifies major and minor device numbers.
events	Identifies the events to check.
reventp	Returns events pointer. This parameter is passed by reference and is used by the **tsselect** entry point to indicate which of the selected events are true at the time of the call.
chan	Specifies the channel number assigned by the **tsmpx** entry point.

Execution Environment

The **tsselect** entry point can be called from the process environment only.

Return Values

The common return codes for the **tsselect** entry point are the following:

ENXIO	Indicates an attempt to use an unconfigured device.
EINVAL	Indicates the select operation was called from a kernel process.
ECHNG	Indicates the channel number is too large.

Related Information

The **tsclose** entry point, **tsconfig** entry point, **tsioctl** entry point, **tsmpx** entry point, **tsopen** entry point, **tsread** entry point, **tswrite** entry point.

The **ddselect** communications PDH entry point.

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

PCI MPQP Device Handler Interface Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications I/O Subsystem: Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications Physical Device Handler Model Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

tswrite Multiprotocol (PCI MPQP) Device Handler Entry Point

Purpose

Provides the means for transmitting data to the IBM ARTIC960Hx PCI (PCI MPQP) device.

Syntax

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

int tswrite (devno, uiop, chan, ext)
dev_t devno;
struct uio *uiop;
int chan, ext;
```

Description

The **tswrite** entry point provides the means for transmitting data to the PCI MPQP device. The kernel calls it when a user-mode process issues a **write** or **writex** subroutine. The **tswrite** entry point can also be called in response to an **fpwrite** kernel service.

The PCI MPQP device handler uses a fixed length buffer for transmitting and receiving data. The maximum supported buffer size is 4096 bytes.

The **tswrite** entry point functions with a 4-Port Multiprotocol Interface adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

tswrite Parameter Block

For the **tswrite** operation, the *ext* parameter points to the **mp_write_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file. The **mp_write_extension** structure contains the following fields:

`status` Identifies the status of the port. The possible values for the returned `status` field are:

CIO_OK
Indicates the operation was successful.

CIO_TX_FULL
Indicates unable to queue any more transmit requests.

CIO_HARD_FAIL
Indicates hardware failure.

CIO_INV_BFER
Indicates invalid buffer (length equals 0, invalid address).

CIO_NOT_STARTED
Indicates device not yet started.

`write_id` Contains a user-supplied correlator. The `write_id` field is returned to the caller by the **CIO_GET_STAT** operation if the **CIO_ACK_TX_DONE** flag is selected in the asynchronous status block.

For a kernel user, this field is returned to the caller with the **stat_fn** function which was provided at open time.

In addition to the common parameters, the **mp_write_extension** structure contains a field for selecting Transparent mode for binary synchronous communication (BSC). Any nonzero value for this field causes Transparent mode to be selected. Selecting Transparent mode causes the adapter to insert data link escape (DLE) characters before all appropriate control characters. Text sent in Transparent mode is unaltered. Transparent mode is normally used for sending binary files.

Note: If an **mp_write_extension** structure is not supplied, Transparent mode can be implemented by the kernel-mode process by imbedding the appropriate DLE sequences in the data buffer.

Parameters

devno Specifies major and minor device numbers.

uiop Points to a **uio** structure that provides variables to control the data transfer operation. The **uio** structure is defined in the **/usr/include/sys/uio.h** file.

chan Specifies the channel number assigned by the **tsmpx** entry point.

ext Specifies the address of the **mp_write_extension** parameter block. If the *ext* parameter is null, no parameter block is specified.

Execution Environment

The **tswrite** entry point can be called from the process environment only.

Return Values

The common return codes for the **tswrite** entry point are the following:

EAGAIN Indicates that the number of direct memory accesses (DMAs) has reached the maximum allowed or that the device handler cannot get memory for internal control structures.
Note: The PCI MPQP device handler does not currently support the **tx_fn** function. If a value of **EAGAIN** is returned by an **tswrite** entry point, the application is responsible for retrying the write.

ECHRNG Indicates that the channel number is too high.

EINVAL Indicates one of the following:

- The port is not set up properly.
- The PCI MPQP device handler could not set up structures for the write.
- The port is not valid.

ENOMEM Indicates that no **mbuf** structure or clusters are available or the total data length is more than a page.

ENXIO Indicates one of the following:

- The port has not been successfully started.
- An invalid adapter number was passed.
- The specified channel number is illegal.

Related Information

The **tsclose** entry point, **tsconfig** entry point, **tsioctl** entry point, **tsmpx** entry point, **tsopen** entry point, **tsread** entry point, **tsselect** entry point.

The **CIO_GET_STAT** (Get Status) tsioctl PCI MPQP Device Handler Operation.

The **write** or **writex** subroutine.

The **uio** structure.

Communications Physical Device Handler Model Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

PCI MPQP Device Handler Interface Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Communications I/O Subsystem: Programming Introduction in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Binary Synchronous Communication (BSC) with the MPQP Adapter in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Sense Data for the Serial Optical Link Device Driver

Note: This information is supported in AIX 5.1 and earlier only.

Sense Data consists of failure data analyzed by the diagnostic programs. The following sense data applies to all the error log entries related to the Serial Optical Link device driver.

Status 1 Register

0x80000000	Program check
0x40000000	Link check
0x20000000	Internal check
0x10000000	Unexpected frame
0x08000000	Reserved bit 4
0x04000000	Connection recovery complete
0x02000000	Connection recovery in progress
0x01000000	Command reject
0x00800000	Secondary command reject
0x00400000	Response time out
0x00200000	Reserved bit 10
0x00100000	Abort sent
0x00080000	Reserved bit 12
0x00040000	Reserved bit 13
0x00020000	Reserved bit 14
0x00010000	Frame discarded
0x00008000	Busy discarded
0x00004000	Reject discarded
0x00002000	Reserved bit 18
0x00001000	Reserved bit 19
0x00000800	Operation complete
0x00000400	Reserved bit 21
0x00000200	Command pending
0x00000100	Primary frame received
0x00000080	Reserved bit 24
0x00000040	Reserved bit 25
0x00000020	Reserved bit 26
0x00000018	One of following:
	0 PU not operational
	1 PU stopped
	2 PU working 1
	3 PU working 2
0x00000004	Reserved bit 28
0x00000003	One of following:
	0 LI connect wait
	1 LI connect try
	2 LI Listen
	3 LI running

Status 2 Register

0x80000000	Receive buffer check
0x40000000	Transmit buffer check
0x20000000	Command check
0x10000000	Synch cmd reject
0x08000000	Reserved bit 4
0x04000000	Tag parity check
0x02000000	Buffer parity check
0x01000000	Storage access check

0x00800000	Reset received
0x00400000	Send count error
0x00200000	Address mismatch
0x00100000	Reserved bit 11
0x00080000	Signal failure
0x00040000	Transmit driver fault
0x00020000	Reserved bit 14
0x00010000	Reserved bit 15
0x00008000	Reserved bit 16
0x00004000	Reserved bit 17
0x00002000	Reserved bit 18
0x00001000	Reserved bit 19
0x00000800	Reserved bit 20
0x00000400	Reserved bit 21
0x00000200	Reserved bit 22
0x00000100	Reserved bit 23
0x00000080	Reserved bit 24
0x00000040	OLS received
0x00000020	NOS received
0x00000010	UD received
0x00000008	UDR received
0x00000004	Reserved bit 29
0x00000002	Signal error
0x00000001	No optics card

Related Information

Status Blocks for the Serial Optical Link Device Driver.

---

sol_close Serial Optical Link Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Resets the Serial Optical Link (SOL) device handler to a known state and frees system resources.

Syntax

```
#include  <sys/device.h>
#include  <sys/comio.h>
#include  <sys/soluser.h>

int sol_close ( devno,  chan)
dev_t devno;
int chan;
```

Parameters

devno	Specifies major and minor device numbers.
chan	Specifies the channel number assigned by the **sol_mpx** entry point.

Description

The **sol_close** entry point is called when a user-mode caller issues a **close** subroutine. The **sol_close** entry point can also be invoked in response to an **fp_close** kernel service.

The **sol_close** entry point functions with a Serial Link Adapter and Serial Optical Channel Converter, that have been correctly configured for use on a qualified network. Consult the hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **sol_close** entry point can be called from the process environment only.

Return Values

ENODEV Indicates that the specified minor number is not valid.

Related Information

The **close** subroutine.

The **fp_close** kernel service.

The **sol_mpx** entry point.

---

sol_config Serial Optical Link Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides functions to initialize and terminate the device handler and to query the Software Vital Product Data (SWVPD).

Syntax

#include <sys/device.h>
#include <sys/uio.h>
#include <sys/comio.h>
#include <sys/soluser.h>
int sol_config (*devno*, *cmd*, *uiop***)**
dev_t *devno*;
int *cmd*;
struct uio **uiop*;

Parameters

devno	Specifies major and minor device numbers.
cmd	Identifies the function to be performed by the **sol_config** routine.
uiop	Points to a **uio** structure that describes the relevant data area for reading or writing.

Description

The **sol_config** entry point is invoked at device configuration time and provides the following operations:

Operation	Description
CFG_INIT	Initializes the Serial Optical Link (SOL) device handler. The device handler registers entry points in the device switch table. The **uio** structure describes the SOL device-dependent structure (DDS) address and length. The device handler copies the DDS into an internal save area.

Operation	Description
CFG_TERM	Terminates the SOL device handler. If there are no outstanding opens, the device handler marks itself terminated and prevents subsequent opens. All dynamically allocated areas are freed. All SOL device handler entry points are removed from the device switch table.
CFG_QVPD	Returns the SOL VPD to the caller. The VPD is placed in the area specified by the caller in the **uio** structure.

The **sol_config** entry point functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **sol_config** entry point can be called from the process environment only.

Return Values for the CFG_INIT Operation

ENOMEM	Indicates the routine was not able to allocate the internal space needed.
EBUSY	Indicates the device was already initialized.
EFAULT	Indicates the specified address is not valid.

Return Values for the CFG_TERM Operation

EBUSY	Indicates there are outstanding opens; not able to terminate.
ENODEV	Indicates there was no device to terminate.

Return Values for the CFG_QVPD Operation

ENODEV	Indicates that there was no device to query the VPD.
EFAULT	Indicates that the specified address is not valid.

Related Information

The **uio** structure in BkSym.TRKernel5;.

---

sol_fastwrt Serial Optical Link Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for kernel-mode users to transmit data to the Serial Optical Link (SOL) device driver.

Syntax

#include <sys/device.h>
#include <sys/comio.h>
#include <sys/soluser.h>
#include <sys/mbuf.h>
int sol_fastwrt (*m*, *chan***)**
struct mbuf ****m*;**
int *chan***;**

Parameters

m Points to an **mbuf** structure containing caller data.
chan Specifies the channel number assigned by the **sol_mpx** entry point.

Description

A kernel user can transmit data more quickly using the **sol_fastwrt** entry point than through a normal **write** system call. The address of the **sol_fastwrt** entry point, along with the *chan* parameter, is given to a kernel-mode caller by way of the **CIO_GET_FASTWRT** sol_ioctl call.

If there is more than one path to the destination, the device handler uses any link that is available. If the **S** (serialized) option was specified on the open, and the connection is point to point, the data is guaranteed to have been received in the order in which it was sent. See the **sol_mpx** entry point for a description of the **S** option.

Note: When communicating through the Network Systems Corp. DX Router, in-order, guaranteed delivery to the destination is not possible. A successful transmission indicates only that the data was successfully received at the DX Router, not necessarily at the final destination. It is the application's responsibility to ensure that the data arrives at the destination.

The data packet must start with a 4-byte field for the destination processor ID (the ID goes in the low-order byte), followed by a 1-byte field for the destination network ID. When the data is received at the destination, the 1-byte processor ID is stripped off, so that the first byte is the 1-byte network ID.

The maximum packet size allowed is **SOL_MAX_XMIT**, as defined in the **/usr/include/sys/soluser.h** file.

The **sol_fastwrt** entry point functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more detailed information on configuring hardware and network qualifications.

Execution Environment

The **sol_fastwrt** entry point can be called from the kernel process environment or the interrupt environment. If the **sol_fastwrt** function is called from the interrupt environment, it is the responsibility of the caller to ensure that the interrupt level is **SOL_OFF_LEVEL**, as defined in the **/usr/include/sys/soluser.h** file, or a less-favored priority.

The **sol_fastwrt** entry point does not support a multiple-packet write. The `m_nextpkt` field in the **mbuf** structure is ignored by the device driver.

The **sol_fastwrt** entry point does not support a write extension. The mbufs are freed when the transmit is complete, and there will be no transmit acknowledgement sent to the caller. If these defaults are not appropriate, use the normal **sol_write** entry point.

The **sol_fastwrt** entry point assumes a trusted caller. The parameter checking done in the normal **sol_write** entry point is not done in **sol_fastwrt**. The caller should ensure such things as a valid channel, page-aligned and page-length mbuf clusters, and a valid packet length.

Return Values

ENODEV Indicates a minor number was specified that was not valid.
ENETDOWN Indicates the network is down. The device is not able to process the write.
ENOCONNECT Indicates the device has not been started.
EAGAIN Indicates the transmit queue is full.
EINVAL Indicates a parameter was specified that was not valid.

ENOMEM	Indicates the device driver was not able to allocate the required memory.
EFAULT	Indicates an invalid address was supplied.
EIO	Indicates an error occurred.

Related Information

The **sol_close** entry point, **sol_config** entry point, **sol_ioctl** entry point, **sol_mpx** entry point, **sol_open** entry point, **sol_read** entry point, **sol_select** entry point, **sol_write** entry point.

The **CIO_GET_FASTWRT** sol_ioctl Serial Optical Link Device Handler Operation.

---

sol_ioctl Serial Optical Link Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides various functions for controlling the Serial Optical Link (SOL) device handler.

Syntax

```
#include  <sys/device.h>
#include  <sys/devinfo.h>
#include  <sys/ioctl.h>
#include  <sys/comio.h>
#include  <sys/soluser.h>

int sol_ioctl ( devno,  cmd,  arg,  devflag,  chan,  ext)
dev_t devno;
int cmd, arg;
ulong devflag;
int chan, ext;
```

Parameters

devno	Specifies major and minor device numbers.
cmd	Specifies the operation to be performed. The possible **sol_ioctl** operation codes are in the **/usr/include/sys/ioctl.h**, **/usr/include/sys/comio.h**, and **/usr/include/sys/soluser.h** files.
arg	Specifies the address of the **sol_ioctl** parameter block.
devflag	Indicates the conditions under which the device was opened.
chan	Specifies the channel number assigned by the **sol_mpx** entry point.
ext	This parameter is not used by the SOL device handler.

Description

The **sol_ioctl** entry point provides various functions for controlling the SOL device handler. The possible **sol_ioctl** operations are:

Operation	Description
CIO_GET_FASTWRT	Provides the attributes of the **sol_fastwrt** entry point.
CIO_GET_STAT	Gets device status.
CIO_HALT	Halts the device.
CIO_QUERY	Queries device statistics.
CIO_START	Starts the device.
IOCINFO	Returns I/O character information.

Operation	Description
SOL_CHECK_PRID	Checks whether a processor ID is connected.
SOL_GET_PRIDS	Gets connected processor IDs.

The **sol_ioctl** entry point functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **sol_ioctl** entry point can be called from the process environment only.

Related Information

The **sol_mpx** entry point.

The **CIO_GET_FASTWRT** sol_ioctl Serial Optical Link Device Handler Operation, **CIO_GET_STAT** sol_ioctl Serial Optical Link Device Handler Operation, **CIO_HALT** sol_ioctl Serial Optical Link Device Handler Operation, **CIO_QUERY** sol_ioctl Serial Optical Link Device Handler Operation, **CIO_START** sol_ioctl Serial Optical Link Device Handler Operation, **IOCINFO** sol_ioctl Serial Optical Link Device Handler Operation, **SOL_CHECK_PRID** sol_ioctl Serial Optical Link Device Handler Operation, **SOL_GET_PRIDS** sol_ioctl Serial Optical Link Device Handler Operation.

CIO_GET_FASTWRT (Get Fast Write) sol_ioctl Serial Optical Link Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the attributes of the **sol_fastwrt** entry point.

Description

The **CIO_GET_FASTWRT** operation provides the attributes of the Serial Optical Link (SOL) device driver's **sol_fastwrt** entry point.

For the **CIO_GET_FASTWRT** operation, the *arg* parameter points to the **cio_get_fastwrt** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Returns one of the following possible status values: • **CIO_OK** • **CIO_INV_CMD**
`fastwrt_fn`	Specifies the function address that can be called to issue a fast path write.
`chan`	Specifies the channel number assigned by the device driver's **mpx** routine.
`devno`	Specifies major and minor device numbers for the device driver, also known as the **dev_t**.

The **CIO_GET_FASTWRT** operation works with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Return Values

EACCES Illegal call from kernel user.
EFAULT Indicates that an address was not valid.
EINVAL Indicates that a parameter was not valid.
ENODEV Indicates that a minor number was not valid.

Related Information

The **sol_fastwrt** entry point, **sol_ioctl** entry point, **sol_write** entry point.

The **CIO_START** sol_ioctl Serial Optical Link Device Handler Operation.

CIO_GET_STAT (Get Status) sol_ioctl Serial Optical Link Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Gets the current status of the Serial Optical Link (SOL) device and device handler.

Description

Note: Only user-mode callers can use the **CIO_GET_STAT** operation.

The **CIO_GET_STAT** operation returns the current status of the SOL device and device handler. For this operation, the *arg* parameter points to a **status_block** structure.

The **CIO_GET_STAT** operation functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Status Blocks for the Serial Optical Link Device Driver

Status blocks contain a code field and possible options. The code field indicates the type of status block (for example, **CIO_START_DONE**). The following are possible status blocks returned by the SOL device driver:

- **CIO_ASYNC_STATUS**
- **CIO_HALT_DONE**
- **CIO_START_DONE**
- **CIO_TX_DONE**

The status block structure is defined in the **/usr/include/sys/comio.h** file and includes the following status codes:

Status Code	Description
code	Indicates one of the following status conditions: • **CIO_ASYNC_STATUS** • **CIO_HALT_DONE** • **CIO_NULL_BLK** • **CIO_START_DONE** • **CIO_TX_DONE**
option[4]	Contains up to four words of additional information, depending on which of the codes listed above is returned.

Status blocks provide status and exception information to users of the SOL device driver.

User-mode processes receive a status block when they request a **CIO_GET_STAT** operation. A user-mode process can wait for the next available status block by issuing a select system call with the specified **POLLPRI** event.

Kernel-mode processes receive a status block by way of the **stat_fn** entry point that is specified at open time.

CIO_ASYNC_STATUS Status Block

The SOL device driver can return the following types of asynchronous status:

- Hard failure status
- Lost data status
- Network Recovery Mode status
- Processor ID status

Hard Failure Status Block Values: When a **CIO_HARD_FAIL** status block is returned, the SOL device is no longer functional. The user should begin shutting down the SOL device driver.

- Unrecoverable Hardware Failure

  When an unrecoverable hardware failure has occurred, the following status block is returned:

Code	CIO_ASYNC_STATUS
option[0]	CIO_HARD_FAIL
option[1]	SOL_FATAL_ERROR
option[2]	Not used
option[3]	Not used

- Exceeded Network Recovery Entry Threshold

  When the SOL device driver has exceeded the entry threshold of the Network Recovery mode, the following status block is returned:

Code	CIO_ASYNC_STATUS
option[0]	CIO_HARD_FAIL
option[1]	SOL_RCVRY_THRESH
option[2]	Not used
option[3]	Not used

Lost Data Status Block Value: For a user-mode process, when the receive queue overflows, the data is lost, and the following status block is returned:

Code	CIO_ASYNC_STATUS
option[0]	CIO_LOST_DATA
option[1]	Not used
option[2]	Not used
option[3]	Not used

Network Recovery Mode Status Block Values:

- Entered Network Recovery Mode

  When the SOL device driver has entered Network Recovery mode, the following status block is returned:

Code	CIO_ASYNC_STATUS
option[0]	CIO_NET_RCVRY_ENTER
option[1]	Not used
option[2]	Not used
option[3]	Not used

- Exited Network Recovery Mode

  When the SOL device driver has exited Network Recovery mode, the following status block is returned:

Code	CIO_ASYNC_STATUS
option[0]	CIO_NET_RCVRY_EXIT
option[1]	Not used
option[2]	Not used
option[3]	Not used

Processor ID Status Block Values:

- New Processor ID

  When the SOL device driver detects a new processor ID that is now reachable, the following status block is returned:

Code	CIO_ASYNC_STATUS
option[0]	SOL_NEW_PRID
option[1]	Indicates the low-order byte contains the new processor ID.
option[2]	Not used.
option[3]	Not used.

- Processor ID Conflict

  When the SOL device driver detects a processor ID conflict, the following status block is returned. The network administrator should ensure that each machine connected to the optical network has a unique processor ID.

Code	CIO_ASYNC_STATUS
option[0]	SOL_PRID_CONFLICT
option[1]	Indicates the low-order byte contains the processor ID that is in conflict.
option[2]	Indicates the low-order byte contains the local processor ID.
option[3]	Not used.

CIO_HALT_DONE Status Block

On a successfully completed **CIO_HALT** operation, the status block is filled as follows:

Code	CIO_HALT_DONE
option[0]	CIO_OK
option[1]	Indicates the low-order bytes are filled in with the netid field passed with the **CIO_START** operation.
option[2]	Not used.
option[3]	Not used.

CIO_START_DONE Status Block

On a successfully completed **CIO_START** operation, the status block is filled as follows:

Code	CIO_START_DONE
option[0]	CIO_OK
option[1]	Indicates the low-order bytes are filled in with the netid field passed with the **CIO_START** operation.
option[2]	Not used.
option[3]	Not used.

If the **CIO_START** operation is unsuccessful, the status block is filled as follows:

Code	CIO_START_DONE
option[0]	Specifies one of the following: • **CIO_TIMEOUT** • **CIO_HARD_FAIL**
option[1]	Indicates the low-order bytes are filled in with the netid field passed with the **CIO_START** operation.
option[2]	Not used.
option[3]	Not used.

CIO_TX_DONE Status Block

When a write request completes for which transmit acknowledgment has been requested, the following status block is built and returned to the caller:

Code	CIO_TX_DONE
option[0]	Specifies one of the following: • **CIO_HARD_FAIL** • **CIO_OK** • **CIO_TIMEOUT**
option[1]	Contains the write_id field specified in the **write_extension** structure in the **write** operation.
option[2]	For a kernel-mode process, contains the **mbuf** pointer that was passed in the **write** operation.
option[3]	Specifies one of the following:
SOL_ACK	Indicates the data was received by the destination processor.
SOL_DOWN_CONN	Indicates the link to the destination has failed.
SOL_NACK_NB	Indicates the destination processor ID cannot allocate enough buffers to receive the data.
SOL_NACK_NR	Indicates the destination processor ID is currently not receiving.
SOL_NACK_NS	Indicates the destination processor ID cannot allocate enough buffers to receive the data.
SOL_NEVER_CONN	Indicates a connection has never been established with the destination processor ID.

Code	CIO_TX_DONE
SOL_NO_CONN	Indicates the destination processor ID is currently not responding.

When the `option[0]` field indicates **CIO_OK**, the data is guaranteed to have been received into memory at the destination. If the **S** (serialized) option was specified on the open, and the connection is point-to-point, the data is guaranteed to have been received in the order in which it was sent.

Note: When communicating through the Network Systems Corp. DX Router, in-order guaranteed delivery to the destination is not possible. A successful transmission indicates only that the data was successfully received at the DX Router, not necessarily at the final destination. It is the application's responsibility to ensure the data arrives at the destination.

Execution Environment

The **CIO_GET_STAT** operation can be called from the process environment only.

Return Values

EACCES	Illegal call from kernel user.
EFAULT	Indicates the specified address is not valid.
EINVAL	Indicates the parameter is not valid.

Related Information

The **sol_ioctl** entry point, **sol_mpx** entry point, **sol_select** entry point.

The **stat_fn** kernel procedure.

The **CIO_START** sol_ioctl Serial Optical Link Device Handler Operation, **CIO_HALT** sol_ioctl Serial Optical Link Device Handler Operation.

CIO_HALT (Halt Device) sol_ioctl Serial Optical Link Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Ends a session with the Serial Optical Link (SOL) device handler.

Description

The **CIO_HALT** operation ends a session with the SOL device handler. The caller indicates the network ID to halt. This **CIO_HALT** operation corresponds to the **CIO_START** operation successfully issued with the specified network ID. A **CIO_HALT** operation should be issued for each **CIO_START** operation successfully issued.

Data for the specified network ID is no longer received. Data received for the specified network ID before the halt is passed to a user-mode caller by the **sol_select** and **sol_read** entry points. Data is passed back to a kernel-mode caller by the **rx_fn** routine specified at open time.

For the **CIO_HALT** operation, the *arg* parameter points to the **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
status	Returns one of the following status values: • **CIO_OK** • **CIO_NETID_INV**
netid	Specifies the network ID. The network ID is placed in the least significant byte of the netid field.

The **CIO_HALT** operation functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **CIO_HALT** operation can be called from the process environment only.

Return Values

EFAULT	Indicates the specified address is not valid.
EINVAL	Indicates the parameter is not valid.
EIO	Indicates a general error. If an extension was provided in the call, additional data identifying the cause of the error can be found in the status field.
ENODEV	Indicates the specified minor number is not valid.

Related Information

Serial Optical Link Device Handler Entry Points.

The **sol_ioctl** entry point, **sol_read** entry point, **sol_select** entry point.

CIO_QUERY (Query Statistics) sol_ioctl Serial Optical Link Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Enables the caller to read the counter values accumulated by the Serial Optical Link (SOL) device handler.

Description

The **CIO_QUERY** operation reads the counter values accumulated by the SOL device handler. The first call to the **sol_open** entry point initializes the counters to 0.

For the **CIO_QUERY** operation, the *arg* parameter points to the **query_parms** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
status	Indicates the status of the command. This field may have a value of **CIO_OK** or **CIO_INV_CMD**.
bufptr	Specifies the address of a buffer where the returned statistics are to be placed.
buflen	Specifies the length of the buffer.
clearall	When the value of this field is **CIO_QUERY_CLEAR**, the counters are cleared upon completion of the call. The **CIO_QUERY_CLEAR** label can be found in the **/usr/include/sys/comio.h** file.

The counters placed in the supplied buffer by the **CIO_QUERY** operation are the counters declared in the **sol_query_stats_t** structure defined in the **/usr/include/sys/soluser.h** file.

The **CIO_QUERY** operation functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult the hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **CIO_QUERY** operation can be called from the process environment only.

Return Values

EFAULT	Indicates the specified address is not valid.
EINVAL	Indicates the parameter is not valid.
EIO	Indicates a general error. If an extension was provided in the call, additional data identifying the cause of the error can be found in the status field.
ENODEV	Indicates the specified minor number is not valid.

Related Information

The **sol_ioctl** entry point, **sol_open** entry point.

CIO_START (Start Device) sol_ioctl Serial Optical Link Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Initiates a session with the Serial Optical Link (SOL) device handler.

Description

The **CIO_START** operation initiates a session with the SOL device handler. If the start is the first on the device, the device handler initializes and opens the SOL. For each successful **CIO_START** call issued, there should be a corresponding **CIO_HALT** operation issued.

After the **CIO_START** operation has successfully completed, the device is ready to transmit and receive data. The caller is free to issue any valid SOL operation. Once started, the adapter receives packets from any of the available optical ports.

The caller notifies the device handler of the network ID to use. The network ID corresponds to the destination service access point (DSAP) in the packet. The caller can issue multiple **CIO_START** operations. The SOL device handler can handle from 0 to the number of network IDs specified by the **SOL_MAX_NETIDS** label. This label is defined in the **/usr/include/sys/soluser.h** file.

For the **CIO_START** operation, the *arg* parameter points to the **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Indicates the status of the **CIO_START**. Possible returned status values are: • **CIO_OK** • **CIO_NETID_FULL** • **CIO_NETID_DUP** • **CIO_NETID_INV**

Field	Description
netid	Specifies the network ID the caller uses on the network. The Network ID is placed in the least significant byte of the netid field. **Note:** Only even number IDs are valid. Odd number IDs are reserved for group IDs not supported for this device and return a status value of **CIO_NETID_INV**

The **CIO_START** operation functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **CIO_START** operation can be called from the process environment only.

Return Values

EADDRINUSE	Indicates the network ID is in use.
EFAULT	Indicates the supplied address is not valid.
EINVAL	Indicates the parameter is not valid.
EIO	Indicates a general error. If an extension was provided in the call, additional data identifying the cause of the error can be found in the status field.
ENETDOWN	Indicates a hardware error for which there is no recovery.
ENODEV	Indicates the specified minor number is not valid.
ENOSPC	Indicates the network ID table is full.

Related Information

The **sol_ioctl** entry point.

The **CIO_HALT** sol_ioctl Serial Optical Link Device Handler Operation.

IOCINFO (Describe Device) sol_ioctl Serial Optical Link Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Returns a structure that describes the Serial Optical Link (SOL) device.

Description

The **IOCINFO** operation returns a structure that describes the SOL device. For this operation, the *arg* parameter points to the **devinfo** structure. This structure is defined in the **/usr/include/sys/devinfo.h** file and contains the following fields:

Field	Description
devtype	Identifies the device type. The SOL device type is **DD_NET_DH**. This value is defined in the **/usr/include/sys/devinfo.h** file.
devsubtype	Identifies the device subtype. The SOL device subtype is **DD_SOL**. This value is defined in the **/usr/include/sys/devinfo.h** file.
broad_wrap	Specifies whether the wrapping of broadcast packets is supported by the device.
rdto	Specifies the configured receive data transfer offset (RDTO) value.
processor_id	Identifies the processor ID used by other systems to address this system. This is a customized attribute in the configuration database.

The parameter block is filled in with the appropriate values upon return.

The **IOCINFO** operation functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **IOCINFO** operation can be called from the process environment only.

Return Values

EFAULT	Indicates the specified address is not valid.
EINVAL	Indicates the parameter is not valid.
ENODEV	Indicates the specified minor number is not valid.

Related Information

The **sol_ioctl** entry point.

SOL_CHECK_PRID (Check Processor ID) sol_ioctl Serial Optical Link Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Checks whether a processor ID is connected to the Serial Optical Link (SOL) subsystem.

Description

The **SOL_CHECK_PRID** operation returns a 0 if the specified processor ID is connected to the SOL subsystem. For this operation, the *arg* parameter is the processor ID to check.

The **SOL_CHECK_PRID** operation functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **SOL_CHECK_PRID** operation can be called from the process environment only.

Return Values

EINVAL	Indicates a parameter is not valid.
ENOCONNECT	Indicates the processor ID is not connected to the SOL subsystem.
ENODEV	Indicates a minor number was specified that is not valid.

Related Information

The **sol_ioctl** entry point.

SOL_GET_PRIDS (Get Processor IDs) sol_ioctl Serial Optical Link Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Returns all processor IDs connected to the Serial Optical Link (SOL) subsystem.

Description

The **SOL_GET_PRIDS** operation returns all processor IDs connected to the SOL subsystem. For this operation, the *arg* parameter points to the **sol_get_prids** structure. This structure is defined in the **/usr/include/sys/soluser.h** file and includes the following fields:

Field	Description
`bufptr`	A pointer to the caller buffer where the list of processor IDs are written. Each processor ID is one byte.
`buflen`	The length of the caller's buffer, in bytes. This is the number of processor IDs the buffer can hold.
`num_ids`	The number of IDs detected. This value is filled in by the SOL device handler. A value greater than the **buflen** value indicates an overflow condition in which there are more processors connected than can be reported in the supplied buffer. If this value is 0, and an error is not returned, no other processor IDs were detected.

The **SOL_GET_PRIDS** operation functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **SOL_GET_PRIDS** operation can be called from the process environment only.

Return Values

EFAULT	Indicates that the specified address is not valid.
EINVAL	Indicates that the parameter is not valid.
EIO	Indicates a general error. If an extension was provided in the call, the `status` field will contain additional data identifying the cause of the error.
ENODEV	Indicates that the minor number specified is not valid.
ENOMEM	Indicates an attempt to get memory failed.

Related Information

The **sol_ioctl** entry point.

sol_mpx Serial Optical Link Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Allocates and deallocates a channel for the Serial Optical Link (SOL) device handler.

Syntax

```
#include  <sys/device.h>
#include  <sys/comio.h>
#include  <sys/soluser.h>

int sol_mpx ( devno,  chanp,  channame)
dev_t devno;
int *chanp;
char *channame;
```

Parameters

devno Specifies major and minor device numbers.

chanp Specifies the channel ID passed as a reference parameter. If the *channame* parameter is null, the *chanp* parameter specifies the ID of the channel to deallocate. Otherwise, this parameter is set to the ID of the allocated channel.

channame Points to the remaining path name describing the channel to allocate. The *channame* parameter accepts the following values:

null Deallocates the channel.

Pointer to a null string
Allows a normal open sequence of the SOL device on the channel ID generated by the **sol_mpx** entry point.

Pointer to a "D"
Allows the SOL device to be opened in Diagnostic mode on the channel ID generated by the **sol_mpx** entry point. Diagnostic mode is only valid when opening a **/dev/op***n* special file.

Pointer to an "F"
Allows a forced open of any of the **/dev/op***n* special files even after the **/dev/ops0** file has been opened.

Pointer to an "S"
Indicates that data serialization is required when the **/dev/ops0** file is being opened. When the Network Systems Corp. DX Router is used for communication, in-order reception cannot be guaranteed.

Description

The **sol_mpx** entry point is not called directly by a user of the SOL device handler. The kernel calls the **sol_mpx** entry point in response to an open or close request.

If the **/dev/ops0** special file is open, the **/dev/op***n* special files cannot be opened unless a forced open is requested. If one or more of the **/dev/op***n* special files are open, opening the **/dev/ops0** special file will succeed, but the ports already opened will not be used. Only one open is allowed for each **/dev/op***n* special file.

The **sol_mpx** entry point functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Note: When the Network Systems Corp. DX Router is used for communication, in-order reception cannot be guaranteed.

Execution Environment

The **sol_mpx** entry point can be called from the process environment only.

Return Values

EPERM Indicates the device is open in a mode that does not allow the Diagnostic-mode open request.
EACCES Indicates a nonprivileged user tried to open the device in Diagnostic mode.
EINVAL Indicates an invalid argument was detected.
EIO Indicates an error occurred.
ENOMEM Indicates memory requests for the open failed.
ENODEV Indicates an invalid minor number was specified.
EBUSY Indicates the maximum number of opens has been exceeded.

Related Information

The **sol_open** entry point.

---

sol_open Serial Optical Link Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Initializes the Serial Optical Link (SOL) device handler and allocates the required system resources.

Kernel-Mode Syntax

```
#include  <sys/device.h>
#include  <sys/comio.h>
#include  <sys/soluser.h>

int sol_open ( devno,  devflag,  chan,  arg)
dev_t devno;
ulong devflag;
int chan;
struct kopen_ext *arg;
```

User-Mode Syntax

```
#include  <sys/device.h>
#include  <sys/comio.h>
#include  <sys/soluser.h>

int sol_open ( devno,  devflag,  chan,  arg)
dev_t devno;
ulong devflag;
int chan;
int arg;
```

Parameters

devno Specifies the major and minor device numbers.
devflag Specifies the flag word with the following definitions:

DKERNEL
Indicates a kernel-mode process. For user-mode processes, this flag must be clear.

DNDELAY
Performs nonblocking reads and writes for this channel. Otherwise, the device handler performs blocking reads and writes for this channel.

chan Specifies the channel number assigned by the **sol_mpx** entry point.

arg Points to a **kopen_ext** structure for kernel-mode processes. The **/usr/include/sys/comio.h** file contains a description of this structure. For user-mode processes, this field is not used.

Description

The **sol_open** entry point is called when a user-mode caller issues an **open**, **openx**, or **creat** subroutine. The **sol_open** routine can also be invoked in response to an **fp_opendev** kernel service. This routine opens a device to read and write data.

The **sol_open** entry point functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Note: After the **sol_open** operation has successfully completed, the caller must issue a **CIO_START** operation before the SOL device handler can transmit or receive any data.

Execution Environment

The **sol_open** entry point can be called from the process environment only.

Return Values

ENODEV Indicates the specified minor number is not valid.
EINVAL Indicates the specified parameter is not valid.
ENOMEM Indicates the device handler was not able to allocate the required memory.
EBUSY Indicates the device is already open in Diagnostic mode.

Related Information

The **sol_mpx** entry point.

The **open**, **openx**, or **creat** subroutine.

The **fp_opendev** kernel service.

The **CIO_START** sol_operation.

---

sol_read Serial Optical Link Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for receiving data from the Serial Optical Link (SOL) device handler.

Syntax

```
#include  <sys/device.h>
#include  <sys/comio.h>
#include  <sys/soluser.h>

int sol_read ( devno,  uiop,  chan,  arg)
dev_t devno;
struct uio *uiop;
int chan;
struct read_extension *arg;
```

Parameters

devno Specifies the major and minor device numbers.

uiop Points to a **uio** structure. For a calling user-mode process, the **uio** structure specifies the location and length of the caller's data area in which to transfer information. The kernel fills in the **uio** structure for the user.

chan Specifies the channel number assigned by the **sol_mpx** entry point.

arg Has a value of null or else points to a **read_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file.

Description

Note: Only user-mode callers should use the **sol_read** entry point.

The **sol_read** entry point provides the means for receiving data from the SOL device handler. When a user-mode caller issues a **read**, **readx**, **readv**, or **readvx** subroutine, the kernel calls the **sol_read** entry point. Any data available for the specified channel is returned.

For this operation, the *arg* parameter may point to the **read_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Contains additional information about the completion of the **sol_read** entry point. Possible values for this field are: **CIO_OK** Indicates the operation was successful. **CIO_BUF_OVRFLW** Indicates the user buffer was too small, and the data was truncated.
`netid`	Not used
`sessid`	Not use.

The data received does contain the 4-byte field for the processor ID. Therefore, the first byte of data will be the `netid` field.

The **sol_read** entry point functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **sol_read** entry point can be called from the process environment only.

Return Values

EACCES	Indicates an illegal call from a kernel-mode user.
ENODEV	Indicates an invalid minor number was specified.
EINTR	Indicates a system call was interrupted.
EMSGSIZE	Indicates the data was too large to fit into the receive buffer and that no *arg* parameter was supplied to provide an alternate means of reporting this error with a status of **CIO_BUF_OVFLW**.
EFAULT	Indicates an invalid address was supplied.
ENOCONNECT	Indicates the device has not been started.

Related Information

Serial Optical Link Device Handler Entry Points.

The **uio** structure in *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1*.

The **sol_mpx** entry point.

The **read**, **readx**, **readv**, or **readvx** subroutine.

---

sol_select Serial Optical Link Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Determines whether a specified event has occurred on the Serial Optical Link (SOL) device.

Syntax

```
#include  <sys/device.h>
#include  <sys/comio.h>
#include  <sys/soluser.h>

int sol_select ( devno,  events,  reventp,  chan)
dev_t devno;
ushort events;
ushort *reventp;
int chan;
```

Parameters

devno Specifies major and minor device numbers.

events Specifies conditions to check, which are denoted by the bitwise OR of one or more of the following:

POLLIN
Check whether receive data is available.

POLLOUT
Check whether transmit available.

POLLPRI
Check whether status is available.

POLLSYNC
Specifies synchronous notification only. The request is not registered for notification on occurrence.

reventp Points to the result of condition checks. A bitwise OR of one of the following conditions is returned:

POLLIN
Receive data is available.

POLLOUT
Transmit available.

POLLPRI
Status is available.

chan Specifies the channel number assigned by the **sol_mpx** entry point.

Description

Note: Only user-mode callers should call this entry point.

The **sol_select** entry point is called when the **select** or **poll** subroutine is used to determine if a specified event has occurred on the SOL device. When the SOL device handler is in a state in which the event can never be satisfied (such as a hardware failure), the **sol_select** entry point sets the returned events flags to 1 (one) for the event that cannot be satisfied. This prevents the **select** or **poll** subroutines from waiting indefinitely.

The **sol_select** entry point functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Execution Environment

The **sol_select** entry point can be called from the process environment only.

Return Values

ENODEV Indicates the specified minor number is not valid.
EACCES Indicates the call from a kernel process is not valid.

Related Information

Serial Optical Link Device Handler Entry Points.

The **sol_mpx** entry point.

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

---

sol_write Serial Optical Link Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for transmitting data to the Serial Optical Link (SOL) device handler.

Syntax

```
#include  <sys/device.h>
#include  <sys/comio.h>
#include  <sys/soluser.h>

int sol_write (devno, uiop, chan, arg)
dev_t devno;
struct uio *uiop;
int chan;
struct write_extension *arg;
```

Parameters

devno Specifies major and minor device numbers.
uiop Points to a **uio** structure specifying the location and length of the caller's data.
chan Specifies the channel number assigned by the **sol_mpx** entry point.
arg Points to a **write_extension** structure. If the *arg* parameter is null, default values are assumed.

Description

The **sol_write** entry point provides the means for transmitting data to the SOL device handler. The kernel calls this entry point when a user-mode caller issues a **write**, **writex**, **writev**, or **writevx** subroutine.

For a user-mode process, the kernel fills in the **uio** structure with the appropriate values. A kernel-mode process must fill in the **uio** structure as described by the **ddwrite** communications entry point.

For the **sol_write** entry point, the *arg* parameter may point to a **write_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Indicates the status condition that occurred. Possible values for the returned status field are: • **CIO_OK** • **CIO_TX_FULL** • **CIO_NOT_STARTED** • **CIO_BAD_RANGE** • **CIO_NOMBUF**
`flag`	Consists of a possible bitwise OR of the following: **CIO_NOFREE_MBUF** Requests that the physical device handler (PDH) not free the **mbuf** structure after transmission is complete. The default is bit clear (free the buffer). For a user-mode process, the PDH always frees the **mbuf** structure. **CIO_ACK_TX_DONE** Requests that when done with this operation, the PDH acknowledges completion by building a **CIO_TX_DONE** status block. In addition, requests the PDH either call the kernel status function or (for a user-mode process) place the status block in the status/exception queue. The default is bit clear (do not acknowledge transmit completion).
`write_id`	For a user-mode caller, the `write_id` field is returned to the caller by the **CIO_GET_STAT** operation (if the **CIO_ACK_TX_DONE** option is selected). For a kernel-mode caller, the `write_id` field is returned to the caller by the **stat_fn** routine that was provided at open time.

The data packet must start with a 4-byte field for the destination processor ID (the ID goes in the low-order byte), followed by a 1-byte field for the destination `netid`. When the data is received at the destination, the 4-byte processor ID will be stripped off, so that the first byte is the 1-byte netid.

The maximum packet size allowed is **SOL_MAX_XMIT**, as defined in the **/usr/include/sys/soluser.h** file.

In case of a link failure, the device handler uses any link that is available. In-order reception of data frames is not guaranteed unless the **S** (serialized) option is specified on the open of the device. See the **sol_mpx** entry point for a description of this option.

The **sol_write** entry point functions with a Serial Link Adapter and Serial Optical Channel Converter that have been correctly configured for use on a qualified network. Consult hardware specifications for more information on configuring hardware and network qualifications.

Note: When the Network Systems Corp. DX Router is used for communication, in-order reception cannot be guaranteed even when using a serialized open.

Execution Environment

The **sol_write** entry point can be called from the process environment only.

Return Values

ENODEV	Indicates the specified minor number is not valid.
ENETDOWN	Indicates the network is down. The device is not able to process the write.
ENOCONNECT	Indicates the device has not been started.
EAGAIN	Indicates the transmit queue is full.
EINVAL	Indicates the specified parameter is not valid.
ENOMEM	Indicates the device handler was not able to allocate the required memory.
EINTR	Indicates a system call was interrupted.
EFAULT	Indicates the address supplied is not valid.

Related Information

The **uio** structure in *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1*.

The **write**, **writex**, **writev**, or **writevx** subroutine.

The **sol_mpx** entry point.

The **stat_fn** routine.

---

tokclose Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Resets the token-ring device handler to a known state and frees system resources.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokclose (devno, chan)
dev_t devno;
int chan;
```

Parameters

devno	Specifies major and minor device numbers.
chan	Identifies the channel number assigned by the **tokmpx** entry point.

Description

The **tokclose** entry point is called when a user-mode caller issues a **close** subroutine. The **tokclose** entry point can also be invoked in response to a **fp_close** kernel service.

The **tokclose** entry point functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **tokclose** entry point can be called from the process environment only.

Return Values

ENXIO Indicates the specified minor number is not valid.

Related Information

The **tokmpx** entry point, **tokopen** entry point.

The **ddclose** Communications PDH entry point.

The **close** subroutine.

The **fp_close** kernel service.

---

tokconfig Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides functions for initializing, terminating, and querying the vital product data (VPD) of the token-ring device handler.

Syntax

```
#include <sys/device.h>
#include <sys/uio.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokconfig
(devno, cmd, uiop)
dev_t devno;
int cmd;
struct uio *uiop;
```

Parameters

devno	Specifies major and minor device numbers.
cmd	Identifies the function to be performed by the **tokconfig** routine.
uiop	Points to a **uio** structure, that describes the relevant data area for reading or writing.

Description

The **tokconfig** entry point provides functions for initializing, terminating, and querying the VPD of the token-ring device handler. The **tokconfig** routine is invoked at device configuration time. The **tokconfig** entry point provides the following three operations:

Operation	Description
CFG_INIT	Initializes the token-ring device handler. The token-ring device handler registers the entry points in the device switch table. The token-ring define device structure (DDS) address and length is described in the **uio** structure. The DDS is copied into an internal save area by the device handler.
CFG_TERM	Terminates the token-ring device handler. If there are no outstanding opens, the token-ring device handler marks itself terminated and prevents subsequent opens. All dynamically allocated areas are freed. All token-ring device handler entry points are removed from the device switch table.

Operation	Description
CFG_QVPD	Returns the token-ring VPD to the caller. The VPD is placed in the area specified by the caller in the **uio** structure.

The **tokconfig** entry point functions with a Token-Ring High Performance Network adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **tokconfig** entry point can be called from the process environment only.

Return Values

Depending on the operation selected, the **tokconfig** entry point returns values.

Return Values for the CFG_INIT Operation

ENOMEM	Indicates the routine was unable to allocate space for the DDS.
EEXIST	Indicates the device was already initialized.
EINVAL	Indicates the DDS provided is not valid.
ENXIO	Indicates the initialization of the token-ring device was unsuccessful.
EFAULT	Indicates that the specified address is not valid.

Return Values for the CFG_TERM Operation

EBUSY	Indicates there are outstanding opens unable to terminate.
ENOENT	Indicates there was no device to terminate.
EACCES	Indicates the device was not configured.
EEXIST	Unable to remove the device from the device switch table.

Return Values for the CFG_QVPD Operation

ENOENT	Indicates there was no device to query the VPD.
EFAULT	Indicates that the specified address is not valid.
EACCES	Indicates the token-ring device handler is not initialized.

Related Information

The **uio** structure in *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1*.

---

tokdump Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for performing a network dump.

Syntax

Refer to the **dddump** entry point for the complete syntax of the dump entry point.

Description

The **tokdump** entry point provides support for six of the seven dump commands. The **DUMPWRITE** command is not supported for network dump. The **tokdumpwrt** entry point supports this write function.

The supported commands are:

DUMPINIT Initializes the token-ring device handler as a dump device.

DUMPQUERY Gets the information required for performing a network dump. The information is returned in the **dmp_query** structure in **/usr/include/sys/dump.h** file. It contains the following information:

- **tokdumpwrt** operation address
- Minimum data transfer size
- Maximum data transfer size

DUMPSTART Starts the network dump processing.

DUMPREAD Initiates a dump read request to the token-ring device handler.

DUMPEND Terminates the network dump processing.

DUMPTERM Terminates the token-ring device hander as a dump device.

The **tokdump** entry point functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **DUMPINIT** command can be called from the process environment only. **DUMPQUERY**, **DUMPSTART**, **DUMPREAD**, **DUMPEND**, and **DUMPTERM** commands can be called in both the process environment and the interrupt environment.

Related Information

The **dddump** entry point, **tokdumpwrt** entry point.

---

tokdumpwrt Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for a network dump program to transmit data to the token-ring device handler.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>
#include <sys/tokuser.h>
#include <sys/mbuf.h>

int tokdumpwrt
(devno, m)
dev_t devno;
struct mbuf *m;
```

Parameters

devno Specifies major and minor device numbers.

m Pointer to an **mbuf** structure containing the data to be transmitted.

Description

The **tokdumpwrt** entry point can be called by a kernel-mode process to pass a write packet to the token-ring device handler for subsequent transmission. The address of this operation is provided to the kernel user by the dump user, who obtains it with the **DUMPQUERY** command.

The **tokdumpwrt** entry point provides for only one data packet to be transmitted for a single **tokdumpwrt** call. The **tokdumpwrt** entry point also assumes that the calling user is a valid kernel user and that the **mbuf** structure contains a valid data packet. It does not free the **mbuf** structure.

The **tokdumpwrt** entry point functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **tokdumpwrt** entry point can be called from the process or interrupt environment.

Return Values

ENODEV Indicates the specified minor number is not valid.
EAGAIN Indicates the transmit queue is full.

Related Information

The **tokdump** entry point, **tokmpx** entry point, **tokopen** entry point.

The Memory Buffer (mbuf) Kernel Services in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

tokfastwrt Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for kernel users to perform direct-access write operations.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>
#include <sys/tokuser.h>
#include <sys/mbuf.h>

int tokfastwrt (devno, m)
dev_t devno;
struct mbuf *m;
```

Parameters

devno Specifies major and minor device numbers.
m Pointer to an **mbuf** structure containing the data to transmit.

Description

The **tokfastwrt** entry point is called from a kernel-mode process to pass a write packet to the token-ring device handler for subsequent transmission. The address of this entry point is provided to the kernel user by the **CIO_GET_FASTWRT** ioctl entry point.

The **tokfastwrt** entry point provides for only one data packet to be transmitted for a single **tokfastwrt** call. The **tokfastwrt** entry point assumes that the calling user is a valid kernel user and that the **mbuf** structure contains a valid data packet. The device handler frees the **mbuf** and does not acknowledge transmit completion.

The **tokfastwrt** entry point functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **tokfastwrt** entry point can be called from a kernel process or interrupt level. The operation level of the token-ring device handler is **TOK_OPLEVEL**. This label is defined in the **/usr/include/sys/tokuser.h** file. The **tokfastwrt** entry point treats this path as a trusted path and the device handler does not check the parameters.

Return Values

ENODEV Indicates the specified minor number is not valid.
EAGAIN Indicates the transmit queue is full.

Related Information

The **tokmpx** entry point, **tokopen** entry point.

The **CIO_GET_FASTWRT** tokioctl Token-Ring Device Handler Operation, **CIO_START** tokioctl Token-Ring Device Handler Operation.

The Memory Buffer (mbuf) Kernel Services in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

tokioctl Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides various functions for controlling the token-ring device handler.

Syntax

```
#include <sys/device.h>
#include <sys/devinfo.h>
#include <sys/ioctl.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokioctl
(devno, cmd, arg, devflag, chan, ext)
dev_t devno;
int cmd, arg;
ulong devflag;
int chan, ext;
```

Parameters

devno	Specifies major and minor device numbers.
cmd	Specifies the operation to be performed. The possible **tokioctl** operation codes can be found in the **/usr/include/sys/ioctl.h**, **/usr/include/sys/comio.h**, and **/usr/include/sys/tokuser.h** files.
arg	Specifies the address of the **tokioctl** parameter block.
devflag	Indicates the conditions under which the device was opened.
chan	Specifies the channel number assigned by the **tokmpx** entry point.
ext	This parameter is not used by the token-ring device handler.

Description

The **tokioctl** entry point provides various functions for controlling the token-ring device handler. The possible **tokioctl** operations are:

Operation	Description
CIO_GET_FASTWRT	Gets function address for the **tokfastwrt** operation.
CIO_GET_STAT	Gets device status.
CIO_HALT	Halts the device.
CIO_QUERY	Queries device statistics.
CIO_START	Starts the device.
IOCINFO	I/O character information.
TOK_FUNC_ADDR	Sets functional addresses.
TOK_GRP_ADDR	Sets the group address.
TOK_QVPD	Queries vital product data (VPD).
TOK_RING_INFO	Queries token-ring information.

The **tokioctl** entry point functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **tokioctl** entry point can be called from the process environment only.

Related Information

The **tokmpx** entry point.

CIO_GET_FASTWRT (Get Fast Write) tokioctl Token-Ring Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the attributes of the **tokfastwrt** entry point.

Description

The **CIO_GET_FASTWRT** tokioctl operation is used to get the parameters required to issue the **tokfastwrt** entry point, which is the kernel-mode fast write command for the token-ring device handler. For the **CIO_GET_FASTWRT** operation, the *arg* parameter points to the **cio_get_fastwrt** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
status	Returns one of the following status values: • **CIO_INV_CMD** • **CIO_OK**
fastwrt_fn	Specifies the address of the **tokfastwrt** entry point.
chan	Specifies the channel ID.
devno	Specifies the major and minor device numbers.

The **CIO_GET_FASTWRT tokioctl** operation functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult the adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **CIO_GET_FASTWRT tokioctl** operation can be called from the kernel-mode process environment only.

Return Values

EFAULT	Indicates that the specified address is not valid, or the calling process is a user-mode process.
EINVAL	Indicates that the specified parameter is not valid.
EIO	Indicates that an error occurred. See the status field for more information.
ENODEV	Indicates that the specified minor number is not valid.
ENXIO	Indicates that an attempt was made to use an unconfigured device.

Related Information

The **tokfastwrt** entry point, **tokioctl** entry point, **tokwrite** entry point.

CIO_GET_STAT (Get Status) tokioctl Token-Ring Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Gets the current status of the token-ring adapter and device handler.

Description

The **CIO_GET_STAT** tokioctl operation returns the current status of the token-ring adapter and device handler. For this operation, the *arg* parameter points to the **status_block** structure. This structure is defined in the **/usr/include/sys/comio.h** file and takes the following status codes:

- **CIO_ASYNC_STATUS**
- **CIO_HALT_DONE**
- **CIO_LOST_STATUS**
- **CIO_NULL_BLK**
- **CIO_START_DONE**
- **CIO_TX_DONE**

Status Blocks for the Token-Ring Device Handler

Status blocks are used to communicate status and exception information to user-mode processes.

User-mode processes receive a status block whenever they request a **CIO_GET_STAT** operation. A user-mode process can wait for the next available status block by issuing a **tokselect** entry point with the specified **POLLPRI** event.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, **CIO_START_DONE**).

There are six possible token-ring status blocks:

- **CIO_ASYNC_STATUS**
- **CIO_HALT_DONE**
- **CIO_LOST_STATUS**
- **CIO_NULL_BLK**
- **CIO_START_DONE**
- **CIO_TX_DONE**

CIO_ASYNC_STATUS Status Block

The token-ring device handler can return the following types of asynchronous status:

- **CIO_HARD_FAIL**
  - **TOK_ADAP_CHECK**
  - **TOK_PIO_FAIL**
  - **TOK_RCVRY_THRESH**
- **CIO_NET_RCVRY_ENTER**
- **CIO_NET_RCVRY_EXIT**
  - **TOK_RING_STATUS**
- **CIO_LOST_DATA**

When a **CIO_HARD_FAIL** status block is returned, the token-ring adapter is no longer functional. The user should shut down the token-ring device handler.

Hard Failure Status Block Values: The following items describe the hard failure status block values for several types of errors.

- Unrecoverable adapter check

  When an unrecoverable adapter check has occurred, this status block is returned:

Code	**CIO_ASYNC_STATUS**
`option[0]`	**CIO_HARD_FAIL**
`option[1]`	**TOK_ADAP_CHECK**
`option[2]`	The adapter return code is in the two high-order bytes. The adapter returns three parameters when an adapter check occurs. Parameter 0 is returned in the two low-order bytes.
`option[3]`	The two high-order bytes contain parameter 1. The two low-order bytes contain parameter 2.

- Unrecoverable PIO error

  When an unrecoverable PIO error has occurred, this status block is returned:

Code	**CIO_ASYNC_STATUS**
`option[0]`	**CIO_HARD_FAIL**
`option[1]`	**TOK_PIO_FAIL**
`option[2]`	Not used
`option[3]`	Not used

- Exceeded network recovery entry threshold

When the token-ring device handler has exceeded the network Recovery mode entry threshold, this status block is returned:

Code	**CIO_ASYNC_STATUS**
option[0]	**CIO_HARD_FAIL**
option[1]	**TOK_RCVRY_THRESH**
option[2]	Not used
option[3]	Not used

Entered Network Recovery Mode Status Block:

When the token-ring device handler has entered network Recovery mode, this status block is returned:

Code	**CIO_ASYNC_STATUS**
option[0]	**CIO_NET_RCVRY_ENTER**
option[1]	Specifies the reason for entering network Recovery mode. Can be one of these seven options: • **TOK_ADAP_CHECK** • **TOK_AUTO_REMOVE** • **TOK_CMD_FAIL** • **TOK_LOBE_WIRE_FAULT** • **TOK_MC_ERROR** • **TOK_REMOVE_RECEIVED** • **TOK_RING_STATUS**
option[2]	Specifies the adapter return code. For an adapter check, the adapter return code is in the two high-order bytes. The adapter returns three parameters when an adapter check occurs. The adapter check parameter 0 is returned in the two low-order bytes.
option[3]	For an adapter check, the two high-order bytes contain parameter 1. The two low-order bytes contain parameter 2.

Exited Network Recovery Mode Status Block:

When the token-ring device handler has exited network Recovery mode, the status block contains the following:

Code	**CIO_ASYNC_STATUS**
option[0]	**CIO_NET_RCVRY_EXIT**
option[1]	Not used
option[2]	Not used
option[3]	Not used

Ring Beaconing Status Block Values:

When the token-ring adapter detects a beaconing condition on the ring, it notifies the device handler. The device handler returns the following status block:

Code	**CIO_ASYNC_STATUS**
option[0]	**TOK_RING_STATUS**
option[1]	**TOK_RING_BEACONING**
option[2]	Specifies the adapter return code. The two low-order bytes contain the ring status.
option[3]	Not used.

Ring Recovered Status Block Values:

When the token-ring detects that the beaconing condition has ceased, it notifies the device handler. The device handler returns the following status block:

Code	**CIO_ASYNC_STATUS**
option[0]	**TOK_RING_STATUS**
option[1]	**TOK_RING_RECOVERED**
option[2]	Not used
option[3]	Not used

Lost Data Status Block: The token-ring device handler has detected lost data due to the receive queue overflowing. The device handler returns the following status block:

Code	**CIO_ASYNC_STATUS**
option[0]	**CIO_LOST_DATA**
option[2]	Not used
option[3]	Not used

CIO_HALT_DONE Status Block

On a successfully completed **CIO_HALT** operation, the status block is filled in as follows:

Code	**CIO_HALT_DONE**
option[0]	**CIO_OK**
option[1]	The two low-order bytes contain the netid field passed with the **CIO_HALT** operation. If a medium access control (MAC) frame session was requested, this field is set to **TOK_MAC_FRAME_NETID**.
option[2]	Not used
option[3]	Not used

CIO_LOST_STATUS Status Block

This status block is returned when it is not available due to a status queue overflow:

Code	**CIO_LOST_STATUS**
option[0]	Not used
option[1]	Not used
option[2]	Not used
option[3]	Not used

CIO_NULL_BLK Status Block

This is returned when the status block is not available.

Code	**CIO_NULL_BLK**
option[0]	Not used
option[1]	Not used
option[2]	Not used
option[3]	Not used

CIO_START_DONE Status Block

On a successfully completed **CIO_START** operation, the following status block is provided:

Code	**CIO_START_DONE**
option[0]	**CIO_OK**
option[1]	The two low-order bytes contain the netid field passed with the **CIO_START** operation. If a MAC frame session was requested, this field is set to **TOK_MAC_FRAME_NETID**.

Code	CIO_START_DONE
option[2]	The two high-order bytes contain the two high-order bytes of the network address. The two low-order bytes are filled in with the 2 middle bytes of the network address.
option[3]	The two high-order bytes contain the two low-order bytes of the network address.

If the **CIO_START** operation is unsuccessful, the status block contains the following:

Code	CIO_START_DONE
option[0]	Can be one of the following options: • **CIO_TIMEOUT** • **TOK_ADAP_CONFIG** • **TOK_ADAP_INIT_FAIL** • **TOK_ADAP_INIT_PARMS_FAIL** • **TOK_ADAP_INIT_TIMEOUT** • **TOK_ADDR_VERIFY_FAIL** • **TOK_LOBE_MEDIA_TST_FAIL** • **TOK_PHYS_INSERT** • **TOK_REQ_PARMS** • **TOK_RING_POLL**
option[1]	The two low-order bytes contain the netid field passed with the **CIO_START** operation. If a MAC frame session was requested, this field is set to **TOK_MAC_FRAME_NETID**.
option[2]	This is the adapter return code. For each of the device-specific codes returned in option[0], an adapter return code is placed in the two low-order bytes of this field. Possible values for the option[2] field are the adapter reset, initialization, and open completion codes.
option[3]	Not used

CIO_TX_DONE Status Block

When a **tokwrite** entry point completes for which transmit acknowledgment has been requested, the following status block is built and returned to the caller.

Code	CIO_TX_DONE
option[0]	**CIO_OK** or **TOK_TX_ERROR**
option[1]	Contains the write_id field specified in the **write_extension** structure passed to the **tokwrite** operation.
option[2]	For a kernel-mode process, contains the mbuf pointer passed in the **tokwrite** operation.
option[3]	The two high-order bytes contain the adapter's transmit command complete code that the adapter returns. The two low-order bytes contain the adapter's transmit CSTAT completion code that is returned when a packet is transmitted by the adapter.

Return Values

EACCES	Indicates an illegal call from a kernel-mode user.
EFAULT	Specifies an address is not valid.
EINVAL	Indicates a parameter is not valid.

Execution Environment

Related Information

The **tokioctl** entry point, **tokopen** entry point, **tokwrite** entry point.

The **CIO_HALT** tokioctl Token-Ring Device Handler Operation, **CIO_START** tokioctl Token-Ring Device Handler Operation.

CIO_HALT (Halt Device) tokioctl Token-Ring Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Ends a session with the token-ring device handler.

Description

The **CIO_HALT** tokioctl operation ends a session with the token-ring device handler. The caller indicates the network ID to halt. This **CIO_HALT** operation corresponds to the **CIO_START** operation successfully issued with the specified network ID. A **CIO_HALT** operation should be issued for each **CIO_START** operation.

Data for the specified network ID is no longer received. Data received for the specified network ID, before the halt, is still passed up to a user-mode caller by **tokselect** and **tokread** entry points. Data is passed back to a kernel-mode caller by the **rx_fn** routine specified at open time.

For the **CIO_HALT** operation, the *arg* parameter points to the **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Returns one of the following status values: • **CIO_NETID_INV** • **CIO_OK**
`netid`	Specifies the network ID. The network ID is placed in the least significant byte of the `netid` field. When terminating the medium-access control (MAC) frame session, the `netid` field should be set to **TOK_MAC_FRAME_NETID**.

The **CIO_HALT** operation functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **CIO_HALT** operation can be called from the process environment only.

Return Values

EFAULT	Indicates the specified address is not valid.
EINVAL	Indicates a parameter is not valid.
ENOMSG	Indicates an error occurred.

Related Information

The **ddioctl** (**CIO_HALT**) operation.

The **CIO_GET_STAT** tokioctl Token-Ring Device Handler Operation, **CIO_START** tokioctl Token-Ring Device Handler Operation.

The **tokselect** entry point, **tokread** entry point, **tokioctl** entry point.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

CIO_QUERY (Query Statistics) tokioctl Token-Ring Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Allows the caller to read the counter values accumulated by a token-ring device handler.

Description

The **CIO_QUERY** tokioctl operation is used by the caller to read the counter values accumulated by a token-ring device handler. The first call to the **tokopen** entry point initializes the counters to 0.

For the **CIO_QUERY** operation, the *arg* parameter points to the **query_parms** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Indicates the status of the port. Returns one of the following status values: • **CIO_OK** • **CIO_INV_CMD**
`buffptr`	Specifies the address of a buffer where the returned statistics are to be placed.
`bufflen`	Specifies the length of the buffer.
`clearall`	When this value equals **CIO_QUERY_CLEAR**, the counters are cleared upon completion of call. The **CIO_QUERY_CLEAR** label can be found in the **/usr/include/sys/comio.h** file.

The counters placed in the supplied buffer by the **CIO_QUERY** operation are the counters declared in the **tok_query_stats_t** structure defined in the **/usr/include/sys/tokuser.h** file.

The **CIO_QUERY** operation functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **CIO_QUERY** operation can be called from the process environment only.

Return Values

EFAULT	Indicates that the specified address is not valid.
EINVAL	Indicates that a parameter is not valid.

Related Information

The **ddioctl** (**CIO_QUERY**) entry point.

The **tokioctl** entry point, **tokopen** entry point.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

CIO_START (Start Device) tokioctl Token-Ring Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Initiates a session with the token-ring device.

Description

The **CIO_START** tokioctl operation initiates a session with the token-ring device handler. If the start is the first on the port, the device handler initializes and opens the token-ring adapter. For each successful **CIO_START** call issued, there should be a corresponding **CIO_HALT** operation issued.

After the **CIO_START** operation has successfully completed, the adapter is ready to transmit and receive data. The caller can issue any valid token-ring operation. Once started, the adapter receives packets that match the token-ring adapter's (hardware) address or the address specified in the device-dependent structure (DDS) and broadcast packets. No group or functional address is specified when the adapter is started.

The caller notifies the device handler which network ID to use. The network ID corresponds to the destination service access point (DSAP) in the token-ring packet. The caller can issue multiple **CIO_START** operations. For each adapter the token-ring device handler can handle from 0 to the number of network IDs specified by the **TOK_MAX_NETIDS** label. This label is defined in the **/usr/include/sys/tokuser.h** file.

The **CIO_START** operation functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

session_blk Parameter Block

For the **CIO_START** operation, the *arg* parameter points to the **session_blk** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Indicates the status of the **CIO_START** operation. Possible returned **status** values are: • **CIO_NETID_DUP** • **CIO_NETID_FULL** • **CIO_OK**
`netid`	Specifies the network ID the caller will use on the network. The network ID is placed in the least significant byte of the `netid` field. To request a medium-access control (MAC) frame session, the `netid` field should be set to the **TOK_MAC_FRAME_NETID** label. This value has a unique identifier in the most significant byte of the `netid` field. There can be only one MAC frame session per adapter.

Note: The token-ring device handler does not allow the caller to specify itself as the wild card network ID.

Execution Environment

The **CIO_START tokioctl** operation can be called from the process environment only.

Return Values

EADDRINUSE	Indicates the network ID is in use.
EINVAL	Indicates a parameter is not valid.

ENETDOWN	Indicates an unrecoverable hardware error.
ENOMSG	Indicates an error.
ENOSPC	Indicates the network ID table is full.

Related Information

The **ddioctl** (**CIO_START**) operation.

The **CIO_GET_STAT** tokioctl Token-Ring Device Handler Operation, **CIO_HALT** tokioctl Token-Ring Device Handler Operation.

The **tokioctl** entry point.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

IOCINFO (Describe Device) tokioctl Token-Ring Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Returns a structure that describes the token-ring device.

Description

The **IOCINFO** tokioctl operation returns a structure that describes the token-ring device. For this operation, the *arg* parameter points to the **devinfo** structure. This structure is defined in the **/usr/include/sys/devinfo.h** file and contains the following fields:

Field	Description
`devtype`	Identifies the device type. The token-ring device type is **DD_NET_DH**. This value is defined in the **/usr/include/sys/devinfo.h** file.
`devsubtype`	Identifies the device subtype. The token-ring device subtype is **DD_TR**. This value is defined in the **/usr/include/sys/devinfo.h** file.
`speed`	Specifies the capabilities of the token-ring device. This is equal to **TOK_4M** when the token-ring device is configured with a data rate of 4 Mbps. The capabilities are **TOK_16M** when the token-ring device is configured with a data rate of 16 Mbps. The **TOK_4M** and **TOK_16M** labels are defined in the **/usr/include/sys/tokuser.h** file.
`broad_wrap`	Specifies whether the wrapping of broadcast packets is supported by the device.
`rdto`	Specifies the configured receive data transfer offset (RDTO) value.
`haddr`	Specifies the 6-byte hardware address of the token-ring adapter card.
`net_addr`	Specifies the 6-byte network address currently used by the token-ring device handler.

The **IOCINFO** operation functions with a Token-Ring High-Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

The parameter block is filled in with the appropriate values upon return.

Execution Environment

The **IOCINFO** tokioctl operation can be called from the process environment only.

Return Values

EFAULT	Indicates the specified address is not valid.
EINVAL	Indicates a parameter is not valid.
ENXIO	Indicates the specified minor number is not valid.

Related Information

The **tokioctl** entry point.

TOK_FUNC_ADDR (Set Functional Address) tokioctl Token-Ring Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Specifies a functional address to be used on a token-ring device.

Description

The **TOK_FUNC_ADDR** tokioctl operation allows the caller to specify a functional address on a token-ring network. A successful **CIO_START** operation must be issued before a **TOK_FUNC_ADDR** operation can be issued. The parameter block for the functional address is the **tok_func_addr_t** structure defined in the **/usr/include/sys/tokuser.h** file.

The **tok_func_addr_t** structure has four fields:

Field	Description
`status`	Returns one of the following status values: • **CIO_INV_CMD** • **CIO_NETID_INV** • **CIO_NOT_STARTED** • **CIO_OK** • **CIO_TIMEOUT**
`netid`	Specifies the network ID associated with this functional address. The network ID must have been successfully started by the **CIO_START** operation. There can only be one functional address specified per network ID.
`opcode`	When set to **TOK_ADD**, the functional address is added to the list of possible functional addresses for which the token-ring adapter accepts packets. When set to **TOK_DEL**, the current functional address is removed from the list of possible functional addresses for which the token-ring adapter accepts packets. The **TOK_ADD** and **TOK_DEL** values are defined in the **/usr/include/sys/tokuser.h** file.
`func_addr`	Specifies the 4 least significant bytes of the 6-byte network function address. The 2 most significant bytes are automatically set to `0xC000` by the token-ring adapter. The most significant bit and the 2 least significant bits within these 4 bytes cannot be set. They are ignored by the token-ring adapter.

The **TOK_FUNC_ADDR** operation functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **TOK_FUNC_ADDR tokioctl** operation can be called from the process environment only.

Return Values

EFAULT	Indicates the specified address is not valid.
EINVAL	Indicates a parameter is not valid.
ENETDOWN	Indicates an unrecoverable hardware error.
ENOCONNECT	Indicates the device has not been started.
ENOMSG	Indicates an error occurred.

Related Information

The **CIO_GET_STAT** tokioctl Token-Ring Device Handler Operation for more information about Token-Ring status blocks.

The **CIO_START** tokioctl Token-Ring Device Handler Operation.

The **tokioctl** entry point.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

TOK_GRP_ADDR (Set Group Address) tokioctl Token-Ring Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Sets the active group address for a token-ring adapter.

Description

The **TOK_GRP_ADDR** tokioctl operation sets the active group address for a token-ring adapter. Only one group address can be specified at a time for a token-ring adapter. For this operation, the *arg* parameter points to the **tok_group_addr_t** structure. This structure is defined in the **/usr/include/sys/tokuser.h** file and contains the following fields:

Field	Description
`status`	Returns one of the following possible status values: • **CIO_INV_CMD** • **CIO_NOT_STARTED** • **CIO_OK** • **CIO_TIMEOUT** • **TOK_NO_GROUP**
`opcode`	When set to **TOK_ADD**, the group address specified is added to the possible address for which the token-ring device accepts packets. When set to **TOK_DEL**, the group address is removed from the possible receive packet addresses. The **TOK_ADD** and **TOK_DEL** values are defined in the **/usr/include/sys/tokuser.h** file.
`group_addr`	Specifies the 4 least significant bytes of the 6-byte network group address. The 2 most significant bytes are automatically set to `0xC000` by the token-ring adapter. The most significant bit within these 4 bytes cannot be set. They are ignored by the token-ring adapter.

The **TOK_GRP_ADDR** operation functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **TOK_GRP_ADDR** tokioctl operation can be called from the process environment only.

Return Values

EFAULT	Indicates the specified address is not valid.
EINVAL	Indicates a parameter is not valid.
ENETDOWN	Indicates an unrecoverable hardware error.
ENOCONNECT	Indicates the device has not been started.
ENOMSG	Indicates an error occurred.

Related Information

Token-Ring Operation Results in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

The **CIO_GET_STAT** tokioctl Token-Ring Device Handler Operation for more information about Token-Ring status blocks.

The **tokioctl** entry point.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

TOK_QVPD (Query Vital Product Data) tokioctl Token-Ring Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Returns the vital product data (VPD) for the token-ring adapter.

Description

The **TOK_QVPD** tokioctl operation returns VPD about the token-ring device. For this operation, the *arg* parameter points to the **tok_vpd_t** block to query the VPD. This structure is defined in the **/usr/include/sys/tokuser.h** file and contains the following fields:

Field	Description
`status`	Returns one of the following status values: • **TOK_VPD_INVALID** • **TOK_VPD_NOT_READ** • **TOK_VPD_VALID**
`l_vpd`	Specifies the length of the *vpd* parameter.
`vpd[TOK_VPD_LENGTH]`	Contains the VPD upon return.

The **TOK_QVPD** operation functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **TOK_QVPD** tokioctl operation can be called from the process environment only.

Return Values

EFAULT Indicates the specified address is not valid.
EINVAL Indicates a parameter is not valid.
ENXIO Indicates the specified minor number is not valid.

Related Information

The **tokioctl** entry point.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

TOK_RING_INFO (Query Token-Ring) tokioctl Token-Ring Device Handler Operation

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Reads information about a token-ring device.

Description

The **TOK_RING_INFO** tokioctl operation reads information about the token-ring device. For this operation, the *arg* parameter points to the **tok_q_ring_info_t** structure. This structure is defined in the **/usr/include/sys/tokuser.h** file and contains the following fields:

`status` Indicates the status condition that occurred. Possible values are:
- **TOK_NO_RING_INFO**
- **CIO_NOT_STARTED**
- **CIO_OK**

`p_info` Points to the buffer where the **tok_ring_info_t** structure is to be copied. The **tok_ring_info_t** structure is defined in the **/usr/include/sys/tokuser.h** file.

`l_buf` Specifies the length of the buffer for the returned ring information structure.

The **TOK_RING_INFO** operation functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **TOK_RING_INFO** operation can be called from the process environment only.

Return Values

EFAULT Indicates a specified address is not valid.
EINVAL Indicates a parameter is not valid.
ENOCONNECT Indicates the device has not been started.
ENOMSG Indicates an error occurred.

Related Information

Token-Ring Operation Results in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

The **CIO_GET_STAT** tokioctl Token-Ring Device Handler Operation for more information about Token-Ring status blocks.

The **tokioctl** entry point.

---

tokmpx Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Allocates and deallocates a channel for the token-ring device handler.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokmpx (devno, chanp, channame)
dev_t devno;
int *chanp;
char *channame;
```

Parameters

devno Specifies major and minor device numbers.

chanp Specifies the channel ID passed as a reference parameter. If the *channame* parameter is null, this is the ID of the channel to be deallocated. Otherwise, this parameter is set to the ID of the allocated channel.

channame Points to the remaining path name describing the channel to allocate. The *channame* parameter accepts the following values:

null Deallocates the channel.

Pointer to a null string
Allows a normal open sequence of the token-ring device on the channel ID generated by the **tokmpx** entry point.

Pointer to a "D"
Allows the token-ring device to be opened in Diagnostic mode on the channel ID generated by the **tokmpx** entry point.

Pointer to a "W"
Allows the token-ring device to be opened in Diagnostic mode with the adapter in Wrap mode on the channel ID generated by the **tokmpx** entry point.

Description

The **tokmpx** entry point is not called directly by a user of the token-ring device handler. The kernel calls the **tokmpx** entry point in response to an open or close request.

If the token-ring device has been successfully opened, any Diagnostic-mode open request is unsuccessful.

The **tokmpx** entry point functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **tokmpx** entry point can be called from the process environment only.

Return Values

EBUSY	Indicates the device was already open in Diagnostic mode and the open request was denied.
ENOMSG	Indicates an error occurred.
ENXIO	Indicates the specified minor number is not valid.
ENOSPC	Indicates the maximum number of opens has been exceeded.

Related Information

The **ddmpx** entry point, **tokclose** entry point, **tokopen** entry point.

The **ddclose** Communications PDH entry point, **ddopen** Communications PDH entry point.

---

tokopen Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Initializes a token-ring device handler and allocates the required system resources.

Syntax

#include <sys/device.h>
#include <sys/comio.h>
#include <sys/tokuser.h>
int tokopen (*devno***,** *devflag***,** *chan***,** *arg***)**
dev_t *devno***;**
ulong *devflag***;**
int *chan***;**
struct kopen_ext * *arg***;**

Parameters

devno Specifies major and minor device numbers.

devflag Specifies the flag word with the following definitions:

DKERNEL
Indicates kernel-mode processes. For user-mode processes, this flag must be clear.

DNDELAY
Specifies that the device handler performs nonblocking reads and writes for this channel. Otherwise, blocking reads and writes are performed for this channel.

chan Specifies the channel number assigned by the **tokmpx** entry point.

arg Points to a **kopen_ext** structure for kernel-mode processes. For user-mode processes, this field is not used.

Description

The **tokopen** entry point is called when a user-mode caller issues an **open**, **openx**, or **creat** subroutine. The **tokopen** routine can also be invoked in response to an **fp_opendev** kernel service. The device is opened to read and write data.

The **tokopen** entry point functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Note: After the **tokopen** operation has successfully completed, the caller must then issue a **CIO_START** operation before any data can be transmitted or received from a token-ring device handler.

Execution Environment

The **tokopen** entry point can be called from the process environment only.

Return Values

ENXIO	Indicates the specified minor number is not valid.
EINVAL	Indicates a specified parameter is not valid.
ENOMEM	Indicates the device handler was unable to allocate the required memory.

Related Information

The **CIO_START** tokioctl Token-Ring Device Handler Operation.

The **open**, **openx** or **creat** subroutine.

The **ddopen** Communications PDH entry point.

The **fp_opendev** kernel service.

---

tokread Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides the means for receiving data from the token-ring device handler.

Syntax

#include <sys/device.h>
#include <sys/uio.h>
#include <sys/comio.h>
#include <sys/tokuser.h>
int tokread (*devno***,** *uiop***,** *chan***,** *arg***)**
dev_t *devno***;**
struct uio * *uiop***;**
int *chan***;**
read_extension * *arg***;**

Parameters

devno	Specifies major and minor device numbers.
uiop	Points to a **uio** structure. For a calling user-mode process, the **uio** structure specifies the location and length of the caller's data area in which to transfer information. The kernel fills in the **uio** structure for the user.
chan	Specifies the channel number assigned by the **tokmpx** entry point.
arg	Can be null or points to the **read_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file.

Description

Note: Only user-mode callers should use the **tokread** entry point.

The **tokread** entry point provides the means for receiving data from the token-ring device handler. When a user-mode caller issues a **read**, **readx**, **readv**, or **readvx** subroutine, the kernel calls the **tokread** entry point.

For this operation, the *arg* parameter may point to the **read_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Contains additional information about the completion of the **tokread** entry point. Possible values for this field are: • **CIO_OK** • **CIO_BUF_OVFLW**
`netid`	Not used
`sessid`	Not used

The **tokread** entry point functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **tokread** entry point can be called from the process environment only.

Return Values

EACCES	Indicates an illegal call from a kernel-mode user.
ENXIO	Indicates the specified minor number is not valid.
EINTR	Indicates a system call was interrupted.
EMSGSIZE	Indicates the data was too large to fit into the receive buffer and that no *arg* parameter was supplied to provide an alternate means of reporting this error with a status of **CIO_BUF_OVFLW**.
EFAULT	Indicates that the supplied address is not valid.
ENOCONNECT	Indicates the device has not been started.

Related Information

The **read**, **readx**, **readv**, or **readvx** subroutine.

The **tokmpx** entry point, **tokwrite** entry point.

Common Communications Status and Exception Codes in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

---

tokselect Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Determines whether a specified event has occurred on the token-ring device.

Syntax

```
#include <sys/device.h>
#include <sys/comio.h>
#include <sys/tokuser.h>

int tokselect ( devno, events, reventp, chan)
dev_t devno;
ushort events;
ushort *reventp;
int chan;
```

Parameters

devno Specifies major and minor device numbers.

events Specifies the conditions to check, denoted by the bitwise OR of one or more of the following:

POLLIN
Check whether receive data is available.

POLLOUT
Check whether transmit available.

POLLPRI
Check whether status is available.

POLLSYNC
Check whether asynchronous notification is available.

reventp Points to the result of condition checks. A bitwise OR one of the following conditions is returned:

POLLIN
Indicates available receive data.

POLLOUT
Indicates available transmit.

POLLPRI
Indicates available status.

chan Specifies the channel number assigned by the **tokmpx** entry point.

Description

Note: Only user-mode callers should call this entry point.

The **tokselect** entry point is called when the **select** or **poll** subroutine is used to determine if a specified event has occurred on the token-ring device.

When the token-ring device handler is in a state in which the event can never be satisfied (for example, an adapter failure), then the **tokselect** entry point sets the returned events flags to 1 for the event that cannot be satisfied. This prevents the **select** or **poll** subroutines from waiting indefinitely.

The **tokselect** entry point functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **tokselect** entry point can only be called from the process environment.

Return Values

ENXIO Indicates the specified minor number is not valid.
EACCES Indicates a call from a kernel process is not valid.

Related Information

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

Select/Poll Logic for ddwrite and ddread Routines.

---

tokwrite Token-Ring Device Handler Entry Point

Purpose

Note: This function is supported in AIX 5.1 and earlier only.

Provides a means of transmitting data to the token-ring device handler.

Syntax

#include <sys/device.h>
#include <sys/uio.h>
#include <sys/comio.h>
#include <sys/tokuser.h>
int tokwrite (*devno***,** *uiop***,** *chan***,** *arg***)**
dev_t *devno***;**
struct uio * *uiop***;**
int *chan***;**
struct write_extension * *arg***;**

Parameters

devno Specifies major and minor device numbers.
uiop Points to a **uio** structure specifying the location and length of the caller's data.
chan Specifies the channel number assigned by the **tokmpx** entry point.
arg Points to a **write_extension** structure. If the *arg* parameter is null, then default values are assumed.

Description

The **tokwrite** entry point provides the means for transmitting data to the token-ring device handler. The kernel calls it when a user-mode caller issues a **write**, **writex**, **writev**, or a **writevx** subroutine.

For a user-mode process, the kernel fills in the **uio** structure with the appropriate values. A kernel-mode process must fill in the **uio** structure as described by the **ddwrite** communications entry point.

For the **tokwrite** entry point, the *arg* parameter may point to a **write_extension** structure. This structure is defined in the **/usr/include/sys/comio.h** file and contains the following fields:

Field	Description
`status`	Indicates the status condition that occurred. Possible values for the returned `status` field are: • **CIO_OK** • **CIO_TX_FULL** • **CIO_NOT_STARTED** • **CIO_NET_RCVRY_MODE**

Field	Description
flag	Consists of a possible bitwise OR one of the following: **CIO_NOFREE_MBUF** Requests that the token-ring device handler not free the **mbuf** structure after transmission is complete. The default is bit clear (free the buffer). For a user-mode process, the token-ring device handler always frees the **mbuf** structure. **CIO_ACK_TX_DONE** Requests that, when done with this operation, the token-ring device handler acknowledges completion by building a **CIO_TX_DONE** status block. In addition, requests the token-ring device handler either call the kernel status function or (for a user-mode process) place the status block in the status/exception queue. The default is bit clear (do not acknowledge transmit completion).
write_id	For a user-mode caller, the write_id field is returned to the caller by the **CIO_GET_STAT** operation (if the **CIO_ACK_TX_DONE** option is selected). For a kernel-mode caller, the write_id field is returned to the caller by the **stat_fn** function that was provided at open time.

The **tokwrite** entry point functions with a Token-Ring High Performance Network Adapter that has been correctly configured for use on a qualified network. Consult adapter specifications for more information on configuring the adapter and network qualifications.

Execution Environment

The **tokwrite** entry point can be called from the process environment only.

Return Values

EAGAIN	Indicates the transmit queue is full.
EFAULT	Indicates an invalid address was supplied.
EINTR	Indicates a system call was interrupted.
EINVAL	Indicates the specified parameter is not valid.
ENETDOWN	Indicates the network is down. The device is unable to process the write.
ENETUNREACH	Indicates the device is in network Recovery mode and unable to process the entry point.
ENOCONNECT	Indicates the device has not been started.
ENOMEM	Indicates the device handler was unable to allocate the required memory.
ENXIO	Indicates the specified minor number is not valid.

Related Information

The **CIO_GET_FASTWRT** tokioctl entry point

The **ddwrite** entry point, **tokfastwrt** entry point, **tokmpx** entry point, **tokopen** entry point.

The **write**, **writex**, **writev**, or **writevx**,subroutine.

The **CIO_START** tokioctl operation.

The **uio** structure in BkSym.TRKernel5;.

See the Use of mbuf Structures in the Communications PDH in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts* for more information about **mbuf** structures.

Chapter 3. LFT Subsystem

lft_t Structure

The **lft_t** structure is defined in the **lft.h** file. The **lft_t** structure is defined as **lft_t** with the **typedef** storage class specifier. The global variable of type **lft_t** is declared within the Low Function Terminal (LFT) subsystem. A pointer to the **lft_t** structure is stored in the **devsw** structure in the LFT device-switch table entry. The **lft_t** structure is defined as follows:

```
typedef struct lft {
       lft_dds_t               *dds_ptr;
       uint                    initialized;
       uint                    open_count;
       unit                    default_cursor;
       struct font_data        *fonts;
       lft_swkbd_t             *swkbd;
       lft_fkp_t               lft_fkp;
       strlft_ptr_t            strlft;
} lft_t, *lft_ptr_t;
```

The **lft_t** structure members are defined as follows:

Structure Member	Description
dds_ptr	Specifies a pointer to the device-dependent structure (DDS). This pointer is initialized by the **lft_init** routine after the DDS has been allocated.
initialized	Specifies a Boolean flag indicating whether LFT is fully initialized.
open_count	Specifies a count of the current number of opens to LFT. When the **open_count** member is decremented to 0, LFT is unconfigured.
default_cursor	Serves as a place holder for a default cursor pointer.
fonts	Specifies a pointer to all of the font information.
swkbd	Specifies a pointer to the software keyboard information.
lft_fkp	Contains font kernel process (**fkproc** attribute) information.
strlft	Specifies streams-specific information.

Related Information

vtmstruct Structure.

phys_displays Structure.

lft_dds_t Structure

The **lft_dds_t** structure is defined in the **lft_dds.h** file and is defined as **lft_dds_t** by the **typedef** storage class specifier. The **lft_dds_t** structure is a common structure that is shared by the Low Function Terminal (LFT) Configure method and the LFT subsystem.

Most of the **lft_dds_t** structure is initialized by the configure method's **build_dds** routine. This routine queries the Object Data Manager (ODM) for all LFT-relevant data. After the **build_dds** routine has completed its initialization of the **lft_dds_t** structure, the configure method calls the **lft_init** routine and passes it the pointer to the **lft_dds_t** structure. The **lft_init** routine then copies the **lft_dds_t** structure from user space into LFT's own local device-dependent structure (DDS) in kernel space. A pointer to this local **lft_dds_t** structure is then stored in the anchored LFT DDS.

The **lft_dds_t** structure contains values initialized by LFT, as well as values from the ODM. The values initialized by LFT are the keyboard file pointer (**kbd.fp**), the display file pointers (**displays**[**i**]**.fp**), and the **vtmstruct** structure pointers (**displays**[**i**]**.vtm_ptr**).

The **lft_dds_t** structure is defined as follows:

```
typedef struct {
        lft_dev_t       lft;
        lft_kbd_t       kbd;
        int             number_of_displays;
        int             default_disp_index;
        char            *swkbd_file;
        char            *font_file_names;
        int             number_of_fonts;
        uint            start_fkproc;
        lft_disp_t      displays[1];
} lft_dds_t;
```

The **lft_dds_t** structure members are defined as follows:

Structure Member	Description

lft	Specifies a structure that contains the device number and logical name of LFT. The **lft** structure is initialized by the LFT Configure method. The **lft** structure is defined as follows:
	`typedef struct {`
	`  dev_t  devno;`
	`  char  devname[NAMESIZE];`
	`} lft_dev_t;`

kbd	Specifies a structure that contains keyboard-specific information. The **kbd** structure is defined as follows:
	`typedef struct {`
	`dev_t  devno;`
	`char  devname[NAMESIZE];`
	`struct file  *fp;`
	`struct diacritic  *diac;`
	`uint  kbd_type;`

number_of_displays	Specifies the total number of displays found to be available by LFT's configure method. This reflects the number of entries in the **lft_disp_info** array.
default_disp_index	Specifies an index into the **displays** array and specifies the display currently in use by LFT. The **default_disp_index** member is initialized by the LFT Configure method. The value of the **default_disp_index** member is set to -1 if the **default_disp** attribute is not found in the ODM. LFT provides an ioctl call that allows the value of the **default_disp_index** member to be changed after LFT has been initialized.
***swkbd_file**	Specifies a pointer to the software-keyboard file name. The LFT Configure method allocates space for the software-keyboard file name. LFT copies the software-keyboard file name into kernel space, opens the file, and reads the software-keyboard information into kernel space.
***font_file_names**	Specifies a pointer to the names of the font files. The LFT Configure method allocates space for the font file names. LFT copies the font file names into kernel space, opens each of the font files, and reads the font information into kernel space. The space allocated in the kernel for holding the font file names is then released.
number_of_fonts	Specifies the number of fonts. The **number_of_fonts** member is initialized by the LFT Configure method.
start_fkproc	Specifies a Boolean flag. This flag is set to True if the LFT Configure method finds an **fkproc** attribute in the ODM for any of the displays associated with LFT. LFT then calls the font server if the flag was set to True.

Structure Member	Description
displays[1]	Specifies an array, the size of which is determined by the number of available displays found during the configuration process. The **displays[1]** structure is defined as follows:

```
typedef struct {
dev_t                   devno;
char                    devname[NAMESIZE];
int                     font_index;
struct file             *fp;
ushort                  fp_valid:
ushort                  flags;
struct vtmstruct        *vtm_ptr;
} lft_disp_t;
```

This is an array of **lft_disp_t** structures, one for each available display. Each structure is tied to a display that has been attached to LFT by the LFT Configure method. The LFT Configure method initializes the device number, device name, and default font index members for each structure associated with an available display. LFT then initializes each **vtmstruct** structure and ***vtm_ptr** file pointer associated with a display. The **number_of_displays** member of the **lft_dds_t** structure defines how many of the **lft_disp_t** structures are valid. The **lft_disp_t** structure members are defined as follows:

devno Specifies the device number of the display adapter. The LFT Configure method initializes this member.

devname{NAMESIZE]
Specifies the logical name of the adapter. The LFT Configure method initializes this member.

font_index
Specifies an integer which contains the index of the default font to be used by the associated adapter. The LFT Configure method initializes this member.

***fp** Specifies a pointer to an integer which specifies the file pointer of the opened display adapter. The ***fp** pointer is used when the display needs to be closed. LFT initializes this member.

fp_valid
Specifies a boolean flag that is set to True if LFT can write to this display. LFT initializes this member.

flags Specifies state flags. Only the **APP_IS_DIAG** flag is currently used.

***vtm_ptr**
Specifies a pointer to a structure of type **vtmstruct**. The ***vtm_ptr** structure pointer is used in all virtual device driver (VDD) calls to the display device driver. LFT allocates and initializes the **vtmstruct** structure.

phys_displays Structure

Each display driver allocates and initializes a **phys_displays** structure during configuration. The **phys_displays** structure is defined in the **/usr/include/sys/display.h** file. The display driver stores a pointer to the **phys_displays** structure in the display driver's **devsw** structure, which is then added to the device switch table. A pointer to the display driver's **vtmstruct** structure is initialized in the **phys_displays** structure when the display driver's **vttact** routine is called. The **phys_displays** structure is defined as follows:

Note: Micro Channel machines only run AIX 5.1 or earlier.

```
struct phys_displays {              /**********************************/
       struct {                     /* data to set up interrupt call */
           struct intr intr;        /* at init time (i_init) */
           long intr_args[4];       /* */
       } interrupt_data;            /**********************************/
       struct phys_displays *same_level; /* other interrupts on same level */
       struct phys_displays *next;  /* ptr to next minor number data */
        struct _gscDev       *pGSC; /* device struct used by rcm */
       dev_t devno;                 /* Device number of this adapter */
       struct lft        *lftanchor;/* lft subsystem */
       int dds_length;              /* length in bytes */
       char *odmdds;                /* ptr to define device structure */
       struct display_info display_info; /* display information */
       uchar disp_devid[4];         /* device information */
                                    /* [1] = 04=display device */
                                    /* [2] = 21=reserved 22=reserved */
                                    /*        25=reserved 27=reserved */
                                    /*        29=reserved */
                                    /* [3] = 00=functional */
                                    /* [4] = 01-04=adapter instance */
       uchar usage;                 /* number of VT's using real screen */
                                    /*   used to prevent deletion of */
                                    /*   real screen from configuration */
                                    /*   if any VT is using it. */
       uchar open_cnt;              /* Open flag for display */
       uchar display_mode;          /* Actual state of the display, */
                                    /* not the virtual terminal: */
                                    /* KSR_MODE or MOM_MODE (see vt.h) */
       uchar dma_characteristics;   /* Attributes related to DMA ops */
#      define DMA_SLAVE_DEV       1 /* Device is bus slave, ow. master */
       struct font_data *default_font; /* Pointer to the default font for */
                                    /*   this display */
       struct vtmstruc       *visible_vt;       /* Pointer to current vt active or */
                                    /*   pseudo-active on THIS display */
                                    /**********************************/
                                    /* DMA Data Areas */
                                    /**********************************/
       int  dma_chan_id;            /* channel id returned from d_init */
       struct dma_bufs              /* DMA buffer structure */
              d_dma_area[MAXDMABUFS]; /* */
                                    /**********************************/
                                    /* Rendering Context Manager Areas */
                                    /**********************************/
       rcmProcPtr cur_rcm;          /* Pointer to current rcm on this */
                                    /* display */
       int num_domains;             /* number of domains */
       int dwa_device;              /* supports direct window access */
       struct _bmr                  /* bus memory ranges */
              busmemr[MAX_DOMAINS]; /* */
       uint io_range;               /* Used for MCA adapter only! */
                                    /* low limit in high short */
                                    /* high limit in low short */
                                    /* to match IOCC register */
       uint *free_area;             /* area free for usage in a device */
                                    /*   dependent manner by the VDD */
                                    /*   for this real screen. */
#ifndef __64BIT_KERNEL
#define RCM_ACC_METHOD_1       (0L) /* MCA and SGA bus adapters */
#endif
#define RCM_ACC_METHOD_2       (1L) /* 60X and PCI bus adapters */
       uint access_method;          /* Access method flags */
#ifndef __64BIT_KERNEL
#define RCM_RUBY_NO_MAP        (1L) /* Tells RCM to not map the space */
#endif
       uint access_flags;           /* Misc flags (used for Ruby now) */
        uint reserved13[13];
       int current_dpm_phase;       /* current phase of DPM this display is in */
```

```
                                /* full-on=1, standby=2, suspend=3, off=4 */
#define       DPMS_ON           0x1
#define       DPMS_STANDBY      0x2
#define       DPMS_SUSPEND      0x3
#define       DPMS_OFF          0x4
       int              NumAddrRanges;
       rcmAddrRange       *AddrRange;
       int reserved4;
       int (*reserved7)();          /**********************************/
                                    /* VDD Function Pointers */
                                    /**********************************/
       int (*vttpwrphase)();        /* power management phase change */
                                    /* function.  It's device dependent */
       int (*vttact)();             /* Activate the display */
       int (*vttcfl)();             /* Move lines around */
       int (*vttclr)();             /* Clear a box on screen */
       int (*vttcpl)();             /* Copy a part of the line */
       int (*vttdact)();            /* Mark the terminal as being */
                                    /*   deactivated */
       int (*vttddf)();             /* Device dependent functions */
                                    /*   i.e. Pacing, context support */
       int (*vttdefc)();            /* Change the cursor shape */
       int (*vttdma)();             /* Issue dma operation */
       int (*vttdma_setup)();       /*  Setup dma */
       int (*vttterm)();            /* Free any resources used */
                                    /*   by this VT */
       int (*vttinit)();            /* setup new logical terminal */
       int (*vttmovc)();            /* Move the cursor to the */
                                    /*   position indicated */
       int (*vttrds)();             /* Read a line segment */
       int (*vtttext)();            /* Write a string of chars */
       int (*vttscr)();             /* Scroll text on the VT */
       int (*vttsetm)();            /* Set mode to KSR or MOM */
       int (*vttstct)();            /* Change color mappings */
       int (*reserved5)();          /* Despite its name, this field is */
                                    /* used for kdb debug */
       int (*bind_draw_read_windows)();
                                    /**********************************/
                                    /* RCM Function Pointers */
                                    /**********************************/
       int (*make_gp)();            /*  Make a graphics process */
       int (*unmake_gp)();          /*  Unmake a graphics process */
       int (*state_change)();       /*  State change handler invoked */
       int (*update_read_win_geom)();
       int (*create_rcx)();         /*  Create a hardware context */
       int (*delete_rcx)();         /*  Delete a hardware context */
#ifdef __64BIT_KERNEL
       int (*reserved21)();
       int (*reserved22)();
       int (*reserved23)();
       int (*reserved24)();
#else
       int (*create_rcxp)();         /*  Create a context part */
       int (*delete_rcxp)();         /*  Delete a context part */
       int (*associate_rcxp)();      /*  Link a part to a context */
       int (*disassociate_rcxp)();   /*  Unlink a part from a context */
#endif
       int (*create_win_geom)();     /*  Create a window on the screen */
       int (*delete_win_geom)();     /*  Delete a window on the screen */
       int (*update_win_geom)();     /*  Update a window on the screen */
#ifdef __64BIT_KERNEL
       int (*reserved25)();
       int (*reserved26)();
       int (*reserved27)();
#else
       int (*create_win_attr)();     /*  Create a window on the screen */
       int (*delete_win_attr)();     /*  Delete a window on the screen */
```

```
       int (*update_win_attr)();     /*  Update a window on the screen */
#endif
       int (*bind_window)();         /*  Update a window bound to rcx */
       int (*start_switch)();        /*  Start a context switch */
                                     /*  Note: This routine runs on */
                                     /*  the interrupt level */
       int (*end_switch)();          /*  Finish the context switch */
                                     /*  started by start_switch() */
#ifdef __64BIT_KERNEL
       int (*reserved28)();
       int (*reserved29)();
       int (*reserved30)();
       int (*reserved31)();
#else
       int (*check_dev)();           /*  Check if this address beints */
                                     /*  to this device. */
                                     /*  Note: this is run on interrupt */
                                     /*  level. */
       int (*async_mask)();          /* Set async events reporting */
       int (*sync_mask)();           /* Set sync events reporting */
       int (*enable_event)();        /* Turns adapter function on */
                                     /* without reports to application */
#endif
       int (*create_thread)();       /* Make a graphics thread */
       int (*delete_thread)();       /* Delete a graphics thread */
       void (*give_up_time_slice)(); /* Relinquish remaining time */
#ifdef __64BIT_KERNEL
       int (*reserved32)();
#else
       int (*diag_svc)();            /* Diagnostics Services (DMA) */
#endif
       int (*dev_init)();            /* Device dep. initialization */
#ifdef __64BIT_KERNEL
       int (*reserved33)();
#else
       int (*dev_term)();            /* Device dep. cleanup */
#endif
                                     /***********************************/
                                     /* Font Support Function Pointers */
                                     /***********************************/
#ifdef __64BIT_KERNEL
        int (*reserved34)();
#else
        int (*pinned_font_ready)();
#endif
       int (*vttddf_fast)();         /* fast ddf functions */
        ushort bus_type;             /* indicates what type of bus */
#ifndef __64BIT_KERNEL
#       define  DISP_BUS_MCA    0x8000/* Microchannel */
#       define  DISP_BUS_SGA    0x4000/* currently not used */
#       define  DISP_BUS_PPC    0x2000/* processor bus */
#       define  DISP_PLANAR     0x0800/* planar registers */
#endif
#       define  DISP_BUS_PCI    0x1000/* PCI bus */

        ushort flags;                /* physical display flags */

#       define GS_DD_DOES_AS_ATT(1L << 0)/* no as_att() by RCM */
                                     /* not currently used */
#       define GS_BUS_AUTH_CONTROL(1L << 1)/* Request bus access ctrl */
#       define GS_HAS_INTERRUPT_HANDLER (1L << 2)/* 1 after i_init() */
                                     /* 0 after i_clear() */
                                     /* not currently used */
#       define GS_DD_SUPPORTS_MP (1L << 3)
```

```
        uint       reserved11[5];     /* not used */
         int             ear;         /* image for EAR reg (xferdata) if !0 */
        uint       spares[18];        /* not used - for future development */
};
```

Related Information

lft Structure.

lft_dds Structure.

vtmstruct Structure.

---

vtmstruct Structure

The **vtmstruct** structure is defined in the **vt.h** file. The Low Function Terminal (LFT) subsystem does not support virtual terminals. However, for backward compatibility with current display drivers, the name of this structure remains the same as in previous releases. The **vtmstruct** structure contains all of the device-dependent data needed by LFT for a given display adapter. LFT allocates and initializes each **vtmstruct** structure. The number of **vtmstruct** structures is determined by the **number_of_displays** variable stored in the **lft_dds** structure. The **vtmstruct** structure is defined as follows:

```
struct vtmstruct {
        struct phys_displays     *display;
        struct vtt_cp_parms      mparms;
        char                     *vttld;
        off_t                    vtid;
        uchar                    vtm_mode;
        int                      font_index;
        int                      number_of_fonts;
        struct font_data         *fonts;
        int                      (*fsp_enq) ();
};
```

The **vtmstruct** structure members are defined as follows:

Structure Member	Description
display	Specifies a pointer to the physical display structure with the display. The ***display** pointer is acquired by LFT by passing the display's device number to the **devswqry** command. The display device drivers initialize the **phys_displays** structures.

mparms	Specifies a structure that contains a code-point mask for implementing 7- or 8-bit ASCII, the code base that is added to the code point if the code base is greater than or equal to 0, the attribute bits, and the cursor position. The *x* and *y* cursor coordinates are initialized to 0. The **vtt_cp_parms** structure is defined as follows:
	`struct vtt_cp_parms`
	`{`
	`ulong cp_mask;`
	`long cp_base;`
	`ushort attributes;`
	`struct vtt_cursor cursor;`
	`};`

vttld	Specifies a pointer to the local data area of the display adapter. The display driver initializes the ***vttld** pointer.
vtid	Specifies the virtual terminal ID. This ID is no longer used, but is retained for backward compatibility. LFT initializes the **vtid** member to 0.
vtm_mode	Specifies a flag which indicates the state of the display. LFT initializes the **vtm_mode** member to ksr mode, and the **vtm_mode** member remains unchanged, since using a hot-key to switch between Keyboard Send-Receive (KSR) and Monitor Mode (MOM) is no longer allowed. The **vtm_mode** member is retained only for backward compatibility.

Structure Member	Description
font_index	Specifies an index into the font structures for a specific font chosen via a **chfont** command. LFT copies this member from the **font_index** member of the **lft_disp_t** structure.
number_of_fonts	Specifies the number of fonts. The **number_of_fonts** member is copied from the **lft_dds** structure during the initialization of the **vtmstruct** structure.

fonts	Specifies a pointer to the array of font tables initialized by LFT. The display driver uses this pointer to acquire its font information.
	
	LFT initializes an array of structures of type **font_data** from data read in from the font files specified in the Object Data Manager (ODM). A pointer to this array is then stored in the **vtmstruct** structure for each display. The display drivers use this pointer to load the appropriate font information. The members of the **font_data** structure are defined as follows:
	
	`struct font_data {`
	`  ulong  font_id;`
	`  char  font_name[20];`
	`  char  font_weight[8];`
	`  char  font_slant[8];`
	`  char  font_page[8];`
	`  ulong  font_style;`
	`  long  font_width;`
	`  long  font_height;`
	`  long  f*font_ptr;`
	`  ulong  font_size;`
	`};`

(*fsq_enq())	Specifies a pointer to the LFT function that queues messages to the font server. LFT initializes this pointer. If a display driver requires the services of the font server, it can queue a message to the font server using the function pointed to by the **(*fsq_enq())** pointer.

Virtual Display Driver (VDD) Interface (lftvi)

Purpose

Provides a communication path from the LFT driver to the lower-level display adapter drivers.

Syntax

```
static int  Function (VP, Down)
struct vtmstruc *VP;
struct down_stream *Down;
```

Description

The **lftvi** interface provides a communication path from the LFT driver to the lower-level display adapter drivers. an array of **vtmstruc** structures with one entry for each configured display adapter is maintained by the **lftvi** interface.

LFT cannot use the normal driver entry points, since the display drivers cannot sleep except in their own open routines. Therefore, all virtual display driver (VDD) functions are called via function pointers in the **phys_display** structure.

The **lftvi** interface includes a collection of functions called by the **vtmupd** and **vtmupd3** subroutines. These functions update information such as cursor position and the tab stop map by calling the appropriate display driver function.

Parameters

Function Specifies one of the functions provided by the **lftvi** interface. The following functions are provided:

cursor_up
Moves the cursor up the number of rows specified in the escape sequence.

cursor_down
Moves the cursor down the number of rows specified in the escape sequence.

cursor_left
Moves the cursor left the number of columns specified in the escape sequence.

cursor_right
Moves the cursor right the number of columns specified in the escape sequence.

cursor_absolute
Moves the cursor to the row and column coordinates specified in the escape sequence.

delete_char
Deletes data from the cursor X position. The number of characters to be deleted is specified in the escape sequence.

delete_line
Deletes the number of lines specified in the escape sequence from the cursor line. Any data following the deleted lines is scrolled up.

erase_l
Erases a line. The escape sequence specifies whether to delete to the end of the line, from the start of the line, or all of the line. This routine calls the **clear_rectangle** function to perform the erasure.

erase_display
Clears all or part of the screen as specified in the escape sequence.

screen_updat
Processes a graphics string. Chops the output string into lines if necessary and calls the **vtt*** routines in the display driver.

copy_part
Calls the VDD that services the terminal to copy part of a line to the presentation space.

clear_rect
Calls the VDD that services the terminal to clear a rectangle.

sound_beep
Calls the sound driver to emit a beep.

set_attributes
Sets the graphics rendition.

update_ds_modes
Sets or resets the data-stream modes.

set_clear_tab
Sets or clears the tabs as specified in the escape sequence. This function operates on either a line or screen model.

update_ht_stop
Sets or clears horizontal tabs. This function can set or clear the horizontal tabs for one line or the whole screen.

clear_all_ht
Clears all horizontal tabs on a line.

cursor_back_tab
Moves the cursor to the previous tab stop.

cursor_ht
Places the cursor at the next horizontal tab.

find_prior_tab
Finds the previous tab by examining the terminal's tab array and setting the cursor's X and Y coordinates to that point. This function takes wrap and autonewline into consideration.

find_next_tab
Finds the next tab by examining the terminal's tab array and setting the cursor's X and Y coordinates to that point. This function takes wrap and autonewline into consideration.

scroll_down
Moves the entire presentation space down the number of lines specified in the escape sequence.

scroll_up
Moves the entire presentation space up the number of lines specified in the escape sequence.

erase_char
Erases the number of characters specified in the escape sequence from the line. If an erase occurs at the end of a line, the line length is altered.

insert_line
Scrolls the cursored line and all lines following it down the number of lines specified in the escape sequence.

insert_char
Inserts the number of empty spaces specified in the escape sequence before the character indicated by the cursor. Characters beginning at the cursor are shifted right. Characters shifted past the right margin are lost.

upd_cursor
Calls the **vttmove** function to update the cursor position.

ascii_index
Moves the cursor down one line. If the cursor was already on the last line, all lines are scrolled up one line.

vttscr Specifies the scroll entry point.

vtttext Specifies the display graphics characters entry point.

vttclr Specifies the clear rectangle entry point.

vttcpl Specifies the copy line entry point.

vttmove
Specifies the move cursor entry point.

vttcfl Specifies the copy full line entry point.

Input Device Driver ioctl Operations

The keyboard special file supports the ioctl operations listed below. Because configuration information is shared between channels, certain ioctl operations such as the **KSTRATE** (set typematic rate) ioctl operation affect both channels regardless of which channel the request is received from.

Operation	Description
IOCINFO	Returns **devinfo** structure.
KSQUERYID	Queries keyboard device identifier.
KSQUERYSV	Queries keyboard service vector.
KSREGRING	Registers input ring.

Operation	Description
KSRFLUSH	Flushes input ring.
KSLED	Illuminates and darkens LEDs on the keyboard.
KSCFGCLICK	Configures the keyboard clicker.
KSVOLUME	Sets alarm volume.
KSALARM	Sounds alarm.
KSTRATE	Sets typematic rate.
KSTDELAY	Sets typematic delay.
KSKAP	Enables/disables keep alive poll.
KSKAPACK	Acknowledges keep alive poll.
KSDIAGMODE	Enables/disables diagnostics mode.
MQUERYID	Queries mouse device identifier.
MREGRING	Registers input ring.
MRFLUSH	Flushes input ring.
MTHRESHOLD	Sets mouse reporting threshold.
MRESOLUTION	Sets mouse resolution.
MSCALE	Sets mouse scale factor.
MSAMPLERATE	Sets mouse sample rate.
TABQUERYID	Queries tablet device identifier.
TABREGRING	Registers input ring.
TABRFLUSH	Flushes input ring.
TABCONVERSION	Sets tablet conversion mode.
TABRESOLUTION	Sets tablet resolution.
TABORIGIN	Sets tablet origin.
TABSAMPLERATE	Sets tablet sample rate.
TABDEADZONE	Sets tablet dead zone.
GIOQUERYID	Queries attached devices.
DIALREGRING	Registers input ring.
DIALRFLUSH	Flushes input ring.
DIALSETGRAND	Sets dial granularity.
LPFKREGRING	Registers input ring.
LPFKRFLUSH	Flushes input ring.
LPFKLIGHT	Sets/resets key lights.

The following ioctl operations are ignored (return immediately with a good return code) when sent to a channel which is not active, and return an **EBUSY** error code if the keyboard is in diagnostics mode:

KSLED

KSCFGCLICK

KSVOLUME

KSALARM

KSTRATE

KSTDELAY

IOCINFO (Return devinfo Structure) ioctl Input Device Driver

Purpose

Returns **devinfo** structure.

Syntax

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

```
int ioctl (FileDescriptor, IOCINFO, Arg)
int FileDescriptor;
struct devinfo *Arg;
```

Description

The **IOCINFO** ioctl operation returns a **devinfo** structure, defined in the **/usr/include/sys/devinfo.h** file, that describes the device. Only the first two fields are valid for this device. The values are as follows:

```
char devtype;   /* device type TBD                */
char flags;     /* open flags (see sys/device.h)  */
```

Parameters

FileDescriptor	Specifies the open file descriptor for the device.
Arg	Specifies the address of the **devinfo** structure.

KSQUERYID (Query Keyboard Device Identifier)

Purpose

Queries keyboard device identifier.

Syntax

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

int ioctl (FileDescriptor, KSQUERYID, Arg)
int FileDescriptor;
uint *Arg;
```

Description

The **KSQUERYID** ioctl subroutine call returns the keyboard device identifier in the location pointed to by the calling argument. Valid keyboard identifiers are:

```
#define   KS101   /0x01  /* 101 keyboard   */
#define   KS102   /0x02  /* 102 keyboard   *
#define   KS106   /0x03  /* 106 keyboard   */
#define   KS101   0x01   /* .......*/
#define   KS102   0x02   /* .......*
#define   KS103   0x03   /* .......*/
```

Parameters

FileDescriptor	Specifies the open file descriptor for the keyboard.
Arg	Specifies the address of the location to return the keyboard identifier.

KSQUERYSV (Query Keyboard Service Vector)

Purpose

Queries keyboard service vector.

Syntax

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

int ioctl (FileDescriptor, KSQUERYSV, Arg)
int FileDescriptor;
caddr_t *Arg;
```

Description

The **KSQUERYSV** ioctl subroutine call returns the address of the keyboard service vector via the calling argument. The keyboard service vector is provided so that certain services may be invoked by kernel extensions without the occurrence of sleeps or page faults. The services provided by the vector must not be invoked by a user process.

The following offsets into the vector are defined:

```
#define   KSVALARM   0   /* sound alarm                          */
#define   KSVSAK     1   /* disable/enable secure attention key  */
#define   KSVRFLUSH  2   /* flush input ring                     */
#define   KSVALARM   0   /*......*/
#define   KSVSAK     1   /*......*/
#define   KSVRFLUSH  2   /*......*/
```

Service vector routines are invoked using an indirect call as follows:

```
(*service_vector[service_number])(dev_t devno, caddr_t arg)
```

where:

- The service vector is a pointer to the service vector obtained by the KSQVERYSU fp_ioctl subroutine call.
- The *service_number* parameter is offset into the service vector.
- The *devno* parameter is the device number for the keyboard.
- The *arg* parameter points to a **ksalarm** structure for alarm requests and an unsigned integer (uint) for secure attention key (SAK) enable/disable requests. The *arg* parameter is NULL for flush queue requests.

A value of zero is returned if the service vector function is successful. Otherwise, an error number defined in the **errno.h** file is returned. Alarm requests are ignored if the kernel extension's channel is not active; enable/disable SAK and queue flush requests are always processed.

The **KSQUERYSV** ioctl subroutine call returns a value of -1 and sets the **errno** global variable to a value of **EINVAL** when called by a user process.

Parameters

FileDescriptor	Specifies the open file descriptor for the keyboard.
Arg	Specifies the address of the location to return the service vector address.

KSREGRING (Register Input Ring)

Purpose

Registers input ring.

Syntax

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

int ioctl (FileDescriptor, KSREGRING, Arg)
int  FileDescriptor;
caddr_t * Arg;
```

Description

If the keyboard special file was opened by a process in user mode, the *Arg* parameter should point to a **uregring** structure containing:

- A pointer to an input ring in user memory.
- The value to be used as the source identifier when enqueuing reports on the ring.
- The size of the input ring in bytes.

If the keyboard special file was opened by a process in kernel mode, the *Arg* parameter should point to a **kregring** structure containing:

- A pointer to an input ring in pinned kernel memory.
- The value to be used as the source identifier when enqueuing reports on the ring.
- A pointer to the notification callback routine. The callback is invoked following the occurrence of an event as specified via the **ir_notify** field in the input ring structure.
- A pointer to the secure attention key (SAK) callback routine. The callback is invoked following the occurrence of a SAK (Ctrl x-r) when SAK detection is enabled.

All callbacks execute within the interrupt environment. All fields within the input ring header as defined by the input ring structure must be properly initialized before the invocation of the ioctl. A subsequent **KSREGRING** ioctl subroutine call replaces the input ring supplied earlier. Specify a null input ring pointer to disable keyboard input.

The input ring acts as a buffer for operator input. Key press and release events are placed on the ring as they occur, without processing or filtering.

Parameters

FileDescriptor	Specifies the open file descriptor for the keyboard.
Arg	Specifies the address of the **uregring** or **kregring** structure.

KSRFLUSH (Flush Input Ring)

Purpose

Flushes input ring.

Syntax

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

int ioctl ( FileDescriptor, KSRFLUSH, NULL)
int FileDescriptor;
```

Description

The **KSRFLUSH** ioctl subroutine call flushes the input ring. The **KSRFLUSH** ioctl subroutine call loads the starting address of the reporting area into the input ring head and tail pointers, then clears the overflow flag.

Parameter

FileDescriptor	Specifies the open file descriptor for the keyboard.

KSLED (Illuminate/Darken Keyboard LEDs)

Purpose

Illuminates and darkens LEDs on the keyboard.

Syntax

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

int ioctl (FileDescriptor, KSLED, Arg)
int  FileDescriptor, * Arg;
```

Description

The **KSLED** ioctl subroutine call illuminates and darkens the LEDs on the natively attached keyboard. The *Arg* parameter points to a bit mask (one bit per LED) that specifies the state of each keyboard LED.

The current state of the keyboard LEDs is returned in the input ring event report for the keyboard.

When keyboard diagnostics are enabled, the **KSLED** ioctl operation fails and sets the **errno** global variable to a value of **EBUSY**.

Parameters

Arg — Specifies the address of the LED bit mask. The bit mask can be any combination of the following values ORed together:

```
#define KSCROLLLOCK   0x01  /*Illuminates ScrollLock LED.*/
#define KSNUMLOCK     0x02  /*Illuminates NumLock LED.*/
#define KSCAPLOCK     0x04  /*Illuminates CapsLock LED.*/
```

FileDescriptor — Specifies the open file descriptor for the keyboard.

KSCFGCLICK (Enable/Disable Keyboard Clicker)

Purpose

Configures the keyboard clicker.

Syntax

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

int ioctl (FileDescriptor, KSCFGCLICK, Arg)
int  FileDescriptor;
uint * Arg;
```

Description

The **KSCFGCLICK** ioctl subroutine call enables and disables the keyboard clicker and sets the clicker's volume. When the keyboard clicker is enabled, the native keyboard speaker generates a sound when a key is pressed.

The **KSCFGCLICK** ioctl subroutine call is supported even when the workstation does not provide a keyboard clicker.

When keyboard diagnostics are enabled, the **KSCFGCLICK** ioctl subroutine call fails and set the **errno** global variable to a value of **EBUSY**.

Parameters

FileDescriptor Specifies the open file descriptor for the keyboard.

Arg Specifies an address of an integer that contains one of the following values:

```
#define KSCLICKOFF   0   /*Turns off clicker.*/
#define KSCLICKLOW   1   /*Sets clicker to low volume.*/
#define KSCLICKMED   2   /*Sets clicker to medium volume.*/
#define KSCLICKHI    3   /*Sets clicker to high volume.*/
```

KSVOLUME (Set Alarm Volume) ioctl

Purpose

Sets alarm volume.

Syntax

```
#include <sys/inputdd.h>int ioctl (FileDescriptor, KSVOLUME, Arg)

int  FileDescriptor;
uint * Arg;
```

Description

The **KSVOLUME** ioctl subroutine call sets the alarm volume.

When keyboard diagnostics are enabled, the **KSVOLUME** ioctl subroutine call fails and sets the **errno** global variable to a value of **EBUSY**.

Parameters

FileDescriptor Specifies the open file descriptor for the keyboard.

Arg Specifies an integer that contains one of the following values:

```
#define KSAVOLOFF   0   /*Turns off alarm.*/
#define KSAVOLLOW   1   /*Sets alarm to low volume.*/
#define KSAVOLMED   2   /*Sets alarm to medium volume*/
#define KSAVOLHI    3   /*Sets alarm to high volume.*/
```

KSALARM (Sound Alarm)

Purpose

Sounds alarm.

Syntax

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

int ioctl (FileDescriptor, KSALARM, Arg)
int  FileDescriptor;
struct ksalarm * Arg;
```

Description

The **KSALARM** ioctl subroutine call causes the native keyboard speaker to produce a sound using the specified frequency and duration. A valid frequency is 32Hz-12KHz inclusive. A valid duration is a number between 0 and 32767. Duration is specified in units of 1/128 of a second, with a maximum of 4.3 minutes.

If the alarm is already on, the request is queued and processed after the previous alarm request has completed. If the queue is full, an **EBUSY** error code is returned. The **KSALARM** function returns immediately if the alarm volume is off (**KSAVOLOFF**) or a duration of 0 is specified.

When keyboard diagnostics are enabled, the **KSALARM** ioctl subroutine call fails and sets the **errno** global variable to a value of **EBUSY**.

Parameters

FileDescriptor	Specifies the open file descriptor for the keyboard.
Arg	Specifies the address of the **KSALARM** structure.

Related Information

The **KSVOLUME** ioctl subroutine call.

The **chhwkbd** command.

KSTRATE (Set Typematic Rate)

Purpose

Sets typematic rate.

Syntax

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

int ioctl (FileDescriptor, KSTRATE, Arg)
int  FileDescriptor;
uint * Arg;
```

Description

The **KSTRATE** ioctl subroutine call changes the rate at which a pressed key repeats itself, specified in number of repeats per second. The minimum rate is 2 repeats per second, and the maximum rate is 30 repeats per second.

When keyboard diagnostics are enabled, the **KSTRATE** ioctl subroutine call fails and sets the **errno** global variable to a value of **EBUSY**.

Parameters

FileDescriptor	Specifies the open file descriptor for the keyboard.
Arg	Specifies the address of an integer that contains the desired typematic rate.

Related Information

The **chhwkbd** command.

KSTDELAY (Set Typematic Delay)

Purpose

Sets typematic delay.

Syntax

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

int ioctl (FileDescriptor, KSTDELAY, Arg)
int  FileDescriptor;
uint * Arg;
```

Description

The **KSTDELAY** ioctl subroutine call sets the time, specified in milliseconds, that a key must be held down before it repeats.

When keyboard diagnostics are enabled, the **KSTDELAY** ioctl subroutine call fails and sets the **errno** global variable to a value of **EBUSY**.

Parameters

FileDescriptor Specifies the open file descriptor for the keyboard.

Arg Specifies the address of a value representing the typematic delay. The *Arg* parameter can be one of the following delay values:

```
#define KSTDLY250   1   250ms.
#define KSTDLY500   2   500ms.
#define KSTDLY750   3   750ms.
#define KSTDLY1000  4  1000ms.
```

Note: For the 106-keyboard, the delays are 300, 400, 500, and 600 milliseconds. All delays are +/- 20%.

Related Information

The **chhwkbd** command.

KSKAP (Enable/Disable Keep Alive Poll)

Purpose

Enables/disables keep alive poll.

Syntax

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

int ioctl (FileDescriptor, KSKAP, Arg)
int  FileDescriptor;
uchar * Arg;
```

Description

The **KSKAP** ioctl subroutine call enables and disables the keep alive poll. The **KSKAP** ioctl subroutine call defines the key sequence that the operator can use to kill the process that owns the keyboard. The *Arg* parameter must point to an array of characters or be equal to NULL. When the *Arg* parameter points to an

array of characters, the first character specifies the number of keys in the sequence. The remainder of the characters in the array define the sequence. Each key of the sequence consists of a position code followed by a modifier flag. The modifier flags can be any combination ok KBDUXSHIFT, KBUXCTRL, and KBDUXALT. If the *Arg* parameter is equal to NULL, the keep alive poll is disabled. A sequence key count of 0 is invalid.

When the keep alive poll is enabled, a **SIGKAP** signal is sent to the user process thatregistered the input ring associated with the active channel when the operator presses and holds down the keys in the order specified by the **KSKAP** ioctl subroutine call. The process must respond with a **KSKAPACK** ioctl subroutine call within 30 seconds or the keyboard driver issues a **SIGKILL** signal to terminate the process.

The keep alive poll is controlled on a per-channel basis and defaults to disabled. The **KSKAP** ioctl subroutine call is not available when the channel is owned by a kernel extension.

Parameters

FileDescriptor	Specifies the open file descriptor for the keyboard.
Arg	Specifies the address of an array of characters or is equal to NULL.

Related Information

The **KSKAPACK** subroutine call.

KSKAPACK (Acknowledge Keep Alive Poll)

Purpose

Acknowledges SIGKAP signals.

Syntax

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

int ioctl (FileDescriptor, KSKAPACK, NULL)
int  FileDescriptor;
```

Description

The **KSKAPACK** ioctl subroutine call acknowledges a **SIGKAP** (keep alive poll) signal.

Parameters

FileDescriptor	Specifies the open file descriptor for the keyboard.

Related Information

The **KSKAP** subroutine call.

KSDIAGMODE (Enable/Disable Diagnostics Mode)

Purpose

Enables/disables diagnostics mode.

Syntax

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

int ioctl (FileDescriptor, KSDIAGMODE, Arg)
uint * Arg;
```

Description

The **KSDIAGMODE** ioctl subroutine call enables and disables keyboard diagnostics mode. When diagnostics mode is enabled, the keyboard driver undefines the keyboard driver interrupt handler and stops processing keyboard events. When diagnostics mode is disabled, the keyboard driver redefines its interrupt handler, then resets and reconfigures the keyboard.

When keyboard diagnostics mode is enabled, the following keyboard ioctl subroutine calls fail and set the **errno** global variable to a value of **EBUSY**:

- **KSLED**
- **KSCFGCLICK**
- **KSVOLUME**
- **KSALARM**
- **KSTRATE**
- **KSTDELAY**

Parameters

FileDescriptor Specifies the open file descriptor for the keyboard.

Arg Specifies the address of an integer that is equal to one of the following values:

```
#define KSDDISABLE   0   /*Disables diagnostics mode.*/
#define KSDENABLE    1   /*Enables diagnostics mode.*/
```

Return Values

The **KSDIAGMODE** ioctl subroutine call returns a value of -1 and sets the **errno** global variable to a value of **EINVAL** when called by a kernel extension. The **KSDIAGMODE** ioctl subroutine call sets the **errno** global variable to a value of **EBUSY** on the RS1/RS2 platform when the tablet special file is open.

MQUERYID (Query Mouse Device Identifier)

Purpose

Queries mouse device identifier.

Syntax

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

int ioctl (FileDescriptor, MQUERYID, Arg)
int FileDescriptor;
unit *Arg;
```

Description

The **MQUERYID** ioctl subroutine call returns the identifier of the natively connected mouse.

Parameters

FileDescriptor Specifies the open file descriptor for the mouse.

Arg Specifies the address of the location to return the mouse identifier. The mouse identifier returned in the *Arg* parameter is:

```
#define     MOUSE3B     0x01     /*.........     */
#define     MOUSE2B     0x02     /*2 Button Mouse*/
```

MREGRING (Register Input Ring)

Purpose

Registers input ring.

Syntax

#include <sys/inputdd.h>
int ioctl (*FileDescriptor***, MREGRING,** *Arg***)**
int *FileDescriptor***;**
struct uregring ****Arg*;**

Description

The **MREGRING** ioctl subroutine call specifies the address of the input ring and the value to be used as the source identifier when enqueuing reports on the ring. A subsequent **MREGRING** ioctl subroutine call replaces the input ring supplied earlier. Specify a null input ring pointer to disable mouse input.

Parameters

FileDescriptor Specifies the open file descriptor for the mouse.

Arg Specifies the address of an **URERING** structure.

MRFLUSH (Flush Input Ring)

Purpose

Flushes input ring.

Syntax

#include <sys/inputdd.h>
int ioctl (*FileDescriptor***, MRFLUSH, NULL)**
int *FileDescriptor***;**

Description

The **MRFLUSH** ioctl subroutine call flushes the input ring. It loads the input ring head and tail pointers with the starting address of the reporting area. The overflow flag is then cleared.

Parameters

FileDescriptor Specifies the open file descriptor for the mouse.

MTHRESHOLD (Set Mouse Reporting Threshold)

Purpose

Sets mouse reporting threshold.

Syntax

#include <sys/inputdd.h>
int ioctl(*FileDescriptor*, **MTHRESHOLD,** *Arg*)
int *FileDescriptor*;
ulong **Arg*;

Description

The **MTHRESHOLD** ioctl subroutine call sets the minimum horizontal or vertical distance (in counts) that the mouse must be moved before the driver reports an event. The high-order two bytes of the *Arg* parameter specify the horizontal threshold and the low-order two bytes specify the vertical threshold. The minimum threshold is 0, while the maximum threshold is 32767. The default horizontal and vertical mouse reporting threshold is 22.

Parameters

FileDescriptor	Specifies the open file descriptor for the mouse.
Arg	Specifies the address of the desired threshold.

MRESOLUTION (Set Mouse Resolution)

Purpose

Sets mouse resolution.

Syntax

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

int ioctl (FileDescriptor, MRESOLUTION, Arg)
int FileDescriptor;
uint *Arg;
```

Description

The **MRESOLUTION** ioctl subroutine call sets the value reported when the mouse is moved one millimeter

Parameters

FileDescriptor	Specifies the open file descriptor for the mouse.
Arg	Specifies the address of an integer where value is one of the following values:

```
#define MRES1     1     /* minimum     */
#define MRES2     2     /*             */
#define MRES3     3     /*             */
#define MRES4     4     /* maximum     */
```

MSCALE (Set Mouse Scale Factor)

Purpose

Sets mouse scale factor.

Syntax

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

int ioctl
(FileDescriptor, MSCALE, Arg)
int  FileDescriptor;
uint * Arg;
```

Description

The **MSCALE** ioctl subroutine call provides a course/fine tracking response. The reported horizontal and vertical movement is converted as follows:

Reported Value

Real Value	1:1 Scale	2:1 Scale
0	0	0
1	1	1
2	2	1
3	3	3
4	4	6
5	5	9
N	N	N x 2
where N >= 6		

The default scale factor is 1:1.

Parameters

FileDescriptor Specifies the open file descriptor for the mouse.

Arg Specifies the address of an integer where value is one of the following values:

```
#define MSCALE11   1  /* 1:1 scale*/
#define MSCALE21   2  /* 2:1 scale*/
```

MSAMPLERATE (Set Mouse Sample Rate)

Purpose

Sets mouse sample rate.

Syntax

#include <sys/inputdd.h> int ioctl (*FileDescriptor***, MSAMPLERATE,** *Arg***)**
int *FileDescriptor***;**
uint **Arg***;*

Description

The **MSAMPLERATE** ioctl subroutine call specifies the maximum number of mouse events that are reported per second.

The default sample rate is 100 samples per second.

Parameters

FileDescriptor Specifies the open file descriptor for the mouse.

Arg Specifies the address of an integer where value is one of the following values:

```
#define MSR10     1     /* 10 samples per second     */
#define MSR20     2     /* 20 samples per second     */
#define MSR40     3     /* 40 samples per second     */
#define MSR60     4     /* 60 samples per second     */
#define MSR80     5     /* 80 samples per second     */
#define MSR100    6     /* 100 samples per second    */
#define MSR200    7     /* 200 samples per second    */
```

TABQUERYID (Query Tablet Device Identifier) ioctl Tablet Device Driver Operation

Purpose

Queries tablet device identifier.

Syntax

#include <sys/inputdd.h>
int ioctl (*FileDescriptor***, TABQUERYID,** *Arg***)**
int *FileDescriptor***;**
struct tabqueryid ****Arg*;**

Description

The **TABQUERYID** ioctl subroutine call returns the identifier of the natively connected tablet and its input device. The first field in the returned structure specifies the model number and may be:

```
#define TAB6093M11   0x01  /* 6093 model 11
or equivalent   */
#define TAB6093M12   0x02  /* 6093 model 12 or equivalent   */
```

The second field in the structure indicates what type of input device is connected to the tablet and may be one of the following:

```
#define TABUNKNOWN  0x00  /* unknown input
device    */
#define TABSTYLUS   0x01  /* stylus                 */
#define TABPUCK     0x02  /* puck                   */
```

Parameters

FileDescriptor Specifies the open file descriptor for the tablet.

Arg Specifies the address of a **TABQUERYID** structure.

TABREGRING (Register Input Ring)

Purpose

Registers input ring.

Syntax

#include <sys/inputdd.h>
int ioctl (*FileDescriptor***, TABREGRING,** *Arg***)**
int *FileDescriptor***;**
struct uregring ****Arg*;**

Description

The **TABREGRING** ioctl subroutine call specifies the address of the input ring and the value to be used as the source identifier when enqueuing reports on the ring. A subsequent **TABREGRING** ioctl subroutine call replaces the input ring supplied earlier. Specify a null input ring pointer to disable tablet input.

Parameters

FileDescriptor	Specifies the open file descriptor for the tablet.
Arg	Specifies the address of a **uregring** structure.

TABRFLUSH (Flush Input Ring

Purpose

Flushes input ring.

Syntax

#include <sys/inputdd.h>
int ioctl (*FileDescriptor***, TABRFLUSH, NULL)**
int *FileDescriptor***;**

Description

The **TABRFLUSH** ioctl subroutine call flushes the input ring. It loads the input ring head and tail pointers with the starting address of the reporting area. The overflow flag is then cleared.

Parameters

FileDescriptor	Specifies the open file descriptor for the tablet.

TABCONVERSION (Set Tablet Conversion Mode)

Purpose

Sets tablet conversion mode.

Syntax

#include <sys/inputdd.h>
int ioctl (*FileDescriptor***, TABCONVERSION,** *Arg***)**
int *FileDescriptor***;**
uint ****Arg*;**

Description

The **TABCONVERSION** ioctl subroutine call specifies whether the value specified by the **TABRESOLUTION** ioctl subroutine call are in English units (inches) or metric units (centimeters).

Parameters

FileDescriptor	Specifies the open file descriptor for the tablet.
Arg	Specifies the address of an integer where value is one of the following values:

```
#define TABINCH 0
/* report coordinates in inches
     */
#define TABCM   1
/* report coordinates in centimeters */
```

Related Information

The **TABRESOLUTION** ioctl subroutine call.

TABRESOLUTION (Set Tablet Resolution)

Purpose

Sets tablet resolution.

Syntax

#include <sys/inputdd.h>
int ioctl (*FileDescriptor***, TABRESOLUTION,** *Arg***)**
int *FileDescriptor***;**
uint ****Arg*;**

Description

The **TABRESOLUTION** ioctl subroutine call specifies the resolution of the tablet in lines per inch. Specify the resolution in lines per inch unless changed by the **TABCONVERSION** ioctl subroutine call. The minimum resolution is 0 and the maximum resolution is 1279 lines per inch or 580 lines per centimeter. The default resolution is 500 lines per inch.

Parameters

FileDescriptor	Specifies the open file descriptor for the tablet.
Arg	Specifies the address of an integer that contains the desired resoultion.

Related Information

The **TABCONVERSION** ioctl subroutine call.

TABORIGIN (Set Tablet Origin)

Purpose
Sets tablet origin.

Syntax
#include <sys/inputdd.h>
int ioctl (*FileDescriptor***, TABORIGIN,***Arg***)**
int *FileDescriptor***;**
uint **Arg***;*

Description
The **TABORIGIN** ioctl subroutine call sets the origin of the tablet to either the lower left-hand corner or the center of the tablet. The default origin is the lower left-hand corner.

Parameters

FileDescriptor	Specifies the open file descriptor for the tablet.
Arg	Specifies the address of an integer whose value is one of the following values:

```
#define TABORGLL     0     /* origin is lower left corner     */
#define TABORGC      1     /* origin is center                */
```

TABSAMPLERATE (Set Tablet Sample Rate) ioctl Tablet Device Driver Operation

Purpose
Sets tablet sample rate.

Syntax
#include <sys/inputdd.h>

int ioctl (*FileDescriptor***, TABSAMPLERATE,** *Arg***)**
int *FileDescriptor***;**
uint **Arg***;*

Description
The **TABSAMPLERATE** ioctl subroutine call specifies the number of times per second that the puck location and button status are sampled. The minimum rate is 0 and the maximum rate is 100. The default rate is one sample per second.

Parameters

FileDescriptor	Specifies the open file descriptor for the tablet.
Arg	Specifies the address of an integer that contains the desired sample rate.

TABDEADZONE (Set Tablet Dead Zone)

Purpose
Sets tablet dead zone.

Syntax

#include <sys/inputdd.h>
int ioctl (*FileDescriptor***, TABDEADZONE,** *Arg***)**
int *FileDescriptor***;**
ulong ****Arg*;**

Description

The **TABDEADZONE** ioctl subroutine call specifies the edges of a zone on the tablet. When the puck is outside of this zone, motion events are not reported (button events are still reported). The high-order two bytes of the *Arg* parameter specify the horizontal edge and the low-order two bytes of the *Arg* parameter specify the vertical edge of the zone. If the tablet is configured with a center origin, the negative of the horizontal value becomes the bottom edge of the zone and the horizontal value becomes the top edge of the zone square. The left and right edges of the zone are generated from the vertical specification in a similar fashion. The minimum horizontal or vertical specification is 0 and the maximum horizontal or vertical specification is 32767.

Parameters

FileDescriptor	Specifies the open file descriptor for the tablet.
Arg	Specifies the address of the dead zone specification.

GIOQUERYID (Query Attached Devices)

Purpose

Queries attached devices.

Syntax

#include <sys/inputdd.h>

int ioctl(*FileDescriptor***, GIOQUERYID,** *Arg***)**
int *FileDescriptor***;**
struct gioqueryid ****Arg*;**

Description

The **GIOQUERYID** ioctl subroutine call returns the identifier of devices connected to the GIO adapter. The ID of the device connected to port 0 is returned in the first field of the structure, and the device connected to port 1 is returned in the second field of the structure. Valid device IDs are as follows:

```
#define giolpfkid   0x01  /* LPFK device ID    */
#define giodialsid  0x02  /* dials device ID   */
```

Parameters

FileDescriptor	Specifies the open file descriptor for the gio adapter.
Arg	Specifies the address of a **gioqueryid** structure.

DIALREGRING (Register Input Ring)

Purpose

Registers input ring.

Syntax

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

int ioctl (FileDescriptor, DIALREGRING, Arg)
int FileDescriptor;
struct uregring *Arg;
```

Description

The **DIALREGRING** ioctl subroutine call specifies the address of the input ring and the value to be used as the source identifier when enqueuing reports on the ring. A subsequent **DIALREGRING** ioctl subroutine call replaces the input ring supplied earlier. Specify a null input ring pointer to disable dial input.

Parameters

FileDescriptor	Specifies the open file descriptor for the dials.
Arg	Specifies the address of the **uregring** structure.

DIALRFLUSH (Flush Input Ring)

Purpose

Flushes input ring.

Syntax

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

int ioctl (FileDescriptor, DIALRFLUSH, Arg)
int FileDescriptor;
```

Description

The **DIALRFLUSH** ioctl subroutine call flushes the input ring. It loads the input ring head and tail pointers with the starting address of the reporting area. The overflow flag is then cleared.

Parameters

FileDescriptor	Specifies the open file descriptor for the dials.

DIALSETGRAND (Set Dial Granularity)

Purpose

Sets dial granularity.

Syntax

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

int ioctl (FileDescriptor, DIALSETGRAND, Arg)
int FileDescriptor;
struct dialsetgrand *Arg;
```

Description

The **DIALSETGRAND** ioctl subroutine call sets the number of events reported per 360 degree revolution, specified as a power of two on a per-dial basis. The **dialsetgrand** structure contains a bit mask that

indicates which dial or dials should be modified. Valid granularity is any number between 2 and 8, inclusive. The default granularity is 7 (128 reports per rotation).

Parameters

FileDescriptor	Specifies the open file descriptor for the dials.
Arg	Specifies the address of the **dialsetgrand** structure.

LPFKREGRING (Register Input Ring)

Purpose

Registers input ring.

Syntax

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

int ioctl (FileDescriptor, LPFKREGRING, Arg)
int FileDescriptor;
struct uregring *Arg;
```

Description

The **LPFKREGRING** ioctl subroutine call specifies the address of the input ring and the value to be used as the source identifier when enqueuing reports on the ring. A subsequent **LPFKREGRING** ioctl subroutine call replaces the input ring supplied earlier. Specify a null input ring pointer to disable LPFK input.

Parameters

FileDescriptor	Specifies the open file descriptor.
Arg	Specifies the address of the **uregring** structure.

LPFKRFLUSH (Flush Input Ring)

Purpose

Flushes input ring.

Syntax

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

int ioctl (FileDescriptor, LPFKRFLUSH, NULL)
int FileDescriptor;
```

Description

The **LPFKRFLUSH** ioctl subroutine call flushes the input ring. It loads the input ring head and tail pointers with the starting address of the reporting area. The overflow flag is then cleared.

Parameters

FileDescriptor	Specifies the open file descriptor.

LPFKLIGHT (Set/Reset Key Lights)

Purpose

Sets/resets key lights.

Syntax

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

int ioctl (FileDescriptor, LPFKLIGHT, Arg)
int FileDescriptor;
ulong *Arg;
```

Description

The **LPFKLIGHT** ioctl subroutine call illuminates and darkens lights associated with keys in the LPFK array. The *Arg* parameter points to a bit mask (one bit per key) that indicates the state (1 = on, 0 = off) of the key's light.

Parameters

FileDescriptor	Specifies the open file descriptor.
Arg	Specifies the address of a bit mask (one bit per key) that indicates the state of the key lights (0 = off, 1 = on).

dd_open LFT Device Driver Interface

Purpose

Allocates device driver resources and ensures exclusive access to a device.

Syntax

```
int dd_open (DevNo, Flag, Chan, Ext)
dev_t  DevNo;
long  Flag,  Chan,  Ext;
```

Description

The **dd_open** low function terminal (LFT) device driver interface allocates resources needed by a device driver and can be used to ensure exclusive access to a device if necessary.

Parameters

DevNo	Specifies the major and minor device numbers.
Flag	Specifies the open file control flags.
Chan	Specifies the channel number (multiplexed devices only).
Ext	Specifies the extension parameter for device-dependent functions.

Return Values

If successful, the **dd_open** device driver interface returns a value of 0. Otherwise, a value of 1 is returned and the **errno** global variable is set to indicate the error.

dd_close LFT Device Driver Interface

Purpose

Deallocates device driver resources and can be used with the **dd_open** low function terminal (LFT) device driver interface to ensure exclusive access to a device.

Syntax

int dd_close (*DevNo***, *Chan*, *Ext***)**
dev_t *DevNo*;
long *Chan*, *Ext*;

Description

The **dd_close** LFT device driver interface deallocates resources used by a device driver and can be used in conjunction with the **dd_open** LFT device driver to ensure exclusive access to a device.

Parameters

DevNo	Specifies the major and minor device numbers.
Chan	Specifies the channel number (multiplexed devices only).
Ext	Specifies the extension parameter for device-dependent functions.

Return Values

If successful, the **dd_close** device driver interface returns a value of 0. Otherwise, a value of 1 is returned and the **errno** global variable is set to indicate the error.

dd_ioctl LFT Device Driver Interface

Purpose

Performs device-dependent processing.

Syntax

int dd_ioctl (*DevNo***, *Cmd*, *Arg*, *DevFlag*, *Chan*, *Ext***)**
dev_t *DevNo*;
long *Cmd*, *Arg*, *DevFlag*, *Chan*, *Ext*;

Description

The **dd_ioctl** low function terminal (LFT) device driver interface performs device-dependent processing not related to reading from and writing to the device.

Parameters

DevNo	Specifies the major and minor device numbers.
Cmd	Specifies the device-dependent command.
Arg	Specifies the command-dependent parameter block address.
DevFlag	Specifies the flag indicating the type of operation.
Chan	Specifies the channel number (multiplexed devices only).
Ext	Specifies the extension parameter for device-dependent functions.

Return Values

If successful, the **dd_ioctl** device driver interface returns a value of 0. Otherwise, a value of 1 is returned and the **errno** global variable is set to indicate the error.

Chapter 4. Printer Subsystems

Subroutines for Print Formatters

The **pioformat** formatter driver provides the following subroutines for the print formatters that it loads, links, and drives:

Subroutine	Description
piocmdout	Outputs an attribute string for a printer formatter.
pioexit	Exits from a printer formatter.
piogetstr	Retrieves an attribute string for a printer formatter.
piogetopt	Used by printer formatters to overlay default flag values from the database with override values from the command line.
piogetvals	Initializes a copy of the database variables for a printer formatter.
piomsgout	Sends a message from a printer formatter.

piocmdout Subroutine

Purpose

Outputs an attribute string for a printer formatter.

Library

None (linked with the **pioformat** formatter driver)

Syntax

```
#include <piostruct.h>

piocmdout (attrname, fileptr, passthru, NULL)
char * attrname;
FILE * fileptr;
int  passthru;
```

Description

The **piocmdout** subroutine retrieves the specified attribute string from the Printer Attribute database and outputs the string to standard output. In the course of retrieval, this subroutine also resolves any logic and any embedded references to other attribute strings or integers.

The *fileptr* and *passthru* parameters are used to pass data that the formatter does not need to scan (for example, graphics data) from the input data stream to standard output.

Parameters

attrname	Points to a two-character attribute name for a string. The attribute name must be defined in the database and can optionally have been defined to the **piogetvals** subroutine as a variable string. The attribute should not be one that has been defined to the **piogetvals** subroutine as an integer.
fileptr	Specifies a file pointer for the input data stream. If the **piocmdout** routine is called from the **lineout** formatter routine, the *fileptr* value should be the *fileptr* passed to the **lineout** routine as a parameter. Otherwise, the *fileptr* value should be **stdin**. If the *passthru* parameter is 0, the *fileptr* parameter is ignored.

passthru Specifies the number of bytes to be passed to standard output unmodified from the input data stream specified by the *fileptr* parameter. This occurs when the **%x** escape sequence is found in the attribute string or in a string included by the attribute string. If no **%x** escape sequence is found, the specified number of bytes is read from the input data stream and discarded. If no bytes are to be passed through, the *passthru* parameter should be 0.

Note: The fourth parameter is reserved for future use. This parameter should be a NULL pointer.

Return Values

Upon successful completion, the **piocmdout** subroutine returns the length of the constructed string.

If the **piocmdout** subroutine detects an error, it issues an error message and terminates the print job.

Related Information

The **lineout** subroutine, **piogetvals** subroutine.

Understanding Embedded References in Printer Attribute Strings in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*

Adding a New Printer Type to Your System in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

---

pioexit Subroutine

Purpose

Exits from a printer formatter.

Library

None (linked with the **pioformat** formatter driver)

Syntax

```
#include <piostruct.h>
void pioexit ( exitcode)
int exitcode;
```

Description

The **pioexit** subroutine should be used by printer formatters to exit either when formatting is complete or an error has been detected. This subroutine is supplied by the formatter driver.

The **pioexit** subroutine has no return values.

Parameters

exitcode Specifies whether the formatting operation completed successfully. A value of **PIOEXITGOOD** indicates that the formatting completed normally. A value of **PIOEXITBAD** indicates that an error was detected.

Related Information

Understanding Embedded References in Printer Attribute Strings in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Adding a New Printer Type to Your System in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

---

piogetattrs Subroutine

Purpose

Retrieves printer attribute values, descriptions, and limits from a printer attribute database.

Library

libqb.a

Syntax

#include <piostruct.h>
int piogetattrs(*QueueName***, *QueueDeviceName*, *NumAttrElems*, *AttrElemTable*)**
const char * *QueueName*, ***** *QueueDeviceName*;
unsigned short *NumAttrElems*;
struct pioattr * *AttrElemTable*;

Description

The **piogetattrs** subroutine retrieves printer attribute values and their associated descriptions and limits from a printer attribute database. Any logic (using the % escape sequence character) within the attribute description will be returned as a text string obtained from a message catalog, and will be in the language determined by the **NLSPATH** and **LANG** environment variables.

Information can be retrieved for any number of attributes defined in the printer attribute database, and for any combination of attribute value, attribute description, and attribute limit for each of the attributes with one **piogetattrs** subroutine call.

The combination of the *QueueName* and *QueueDeviceName* parameters identify a specific printer attribute database. Therefore, the *QueueName* and *QueueDeviceName* parameters must be unique for a particular host.

Parameters

QueueName	Specifies the print queue name. The print queue does not have to exist.
QueueDeviceName	Specifies the queue device name for the print queue name specified by the *QueueName* parameter. The queue device does not have to exist.
NumAttrElems	Specifies the number of attribute elements in the table specified by the *AttrElemTable* parameter.
AttrElemTable	Points to a table of attribute element structures. Each structure element in the table specifies an attribute name, the type of value to be returned for the attribute, fields where the location and length of the returned value are to be stored, and a field for the return code of the retrieval operation. Memory is allocated for each resolved value that is returned, and the memory location and length are returned in the structure element. The format of each structure element is defined by the **pioattr** structure definition in the **/usr/include/piostruct.h** file.

Return Values

NumAttrElems	Specifies the number of attribute elements for which the **piogetattrs** subroutine has successfully retrieved the requested information.
-1	Indicates that an error occurred.

Examples

```
/* Array of elements to be passed to
piogetattrs() */
#define ATTR_ARRAY_NO (sizeof(attr_table)/sizeof(attr_table[0]))

struct pioattr attr_table[] = {
        {"_b", PA_AVALT, NULL, 0, 0}, /* attribute record      */
                                      /* for _b (bottom margin)*/
        {"_i", PA_AVALT, NULL, 0, 0}, /* attribute record for  */
                                      /* _i (left indentation) */
        {"_t", PA_AVALT, NULL, 0, 0}, /* attribute record for  */
                                      /* _t (top margin)       */
}

...
const char                      *qnm = "ps";
const char                      *qdnm = "lp0";
int                             retno;
register const pioattr_t        *pap;

...
if((retno = piogetattrs(qnm,qdnm,ATTR_ARRAY_NO,attr_table)) ==-1)          {(void)
fprintf(stderr,"Fatal error in piogetattrs()\n");
...
}
else if (retno != ATTR_ARRAY_NO) _{
        (void) printf("Warning! Infor was not retrieved for all \
        the attributes.\n");
}
for(pap = attr_table; pap<attr_table+ATTR_ARRAY_NO;pap++)
        if(pap->pa_retcode) /* If info was successfully */
                            /* retrieved for this attr  */
...
```

---

piogetopt Subroutine

Purpose

Overlays default flag values from the database colon file with override values from the command line.

Library

None (linked with the **pioformat** formatter driver)

Syntax

```
#include <piostruct.h>

int piogetopt ( argc,  argv, NULL, NULL)
int argc;
char *argv [];
```

Description

The **piogetopt** subroutine should be used by a printer formatter's **setup** routine to perform these three tasks:

- Parse the command line flags.
- Convert the flag arguments, as needed, to the data types specified in the array of **attrparms** structures previously passed to the **piogetvals** subroutine.
- Overlay the default flag arguments with values from the database.

The **piogetopt** subroutine is supplied by the formatter driver.

The database attribute names for flags with integer arguments must have previously been defined to the formatter driver with the **piogetvals** subroutine. Based on the information that was provided to the **piogetvals** subroutine, the **piogetopt** subroutine takes these three actions:

- Recognizes each flag argument that needs to be converted to an integer value.
- Converts the argument string to an integer value using the conversion method specified to the **piogetvals** subroutine.
- Regardless of the data type (integer variable, string variable, or string constant), overlays the default value from the database.

Parameters

argc Same as the *argc* parameter received by the formatter's **setup** routine when it was called by the formatter driver.

argv Same as the *argv* parameter received by the formatter's **setup** routine when it was called by the formatter driver.

Note: The third parameter, NULL, is a place holder. The fourth parameter, NULL, is reserved for future use. The fourth parameter should be a NULL pointer.

Return Values

A return value of 0 indicates successful completion. If the **piogetopt** subroutine detects an error, it issues an error message and terminates the print job.

Related Information

The **piogetvals** subroutine, **setup** subroutine.

Understanding Embedded References in Printer Attribute Strings in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Adding a New Printer Type to Your System in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

---

piogetstatus Subroutine

Purpose

Retrieves print job status information from a status file.

Library

libqb.a

Syntax

```
#include <IN/stfile.h>

int piogetstatus(StatusFileDescriptor,
VersionMagicNumber, StatusInformation)
int StatusFileDescriptor, VersionMagicNumber;
void *StatusInformation;
```

Description

The information returned by the **piogetstatus** subroutine includes the queue name, queue device name, job number, job status, percent done, and number of pages printed. The **piogetstatus** subroutine reads the specified status file and places the information in the structure specified by the *StatusInformation* parameter. The format of the status structure is determined by the version magic number specified by the *VersionMagicNumber* parameter. Each time there is a change in the status file structure for a new release, a unique number is assigned to the release's version magic number. This supports structure formats of previous releases.

Parameters

StatusFileDescriptor	Specifies the file descriptor of the status file. The *StatusFileDescriptor* parameter must specify a value of 3, because the print spooler always opens a status file with a file descriptor value of 3.
VersionMagicNumber	Specifies the version magic number that identifies the format of the status structure in which information is specified.
StatusInformation	Specifies a generic pointer to a status structure that contains print job status information that is to be stored in the status file.

Return Values

1 Indicates that the **pioputstatus** subroutine was successful.
-1 Indicates that an error occurred.

---

piogetstr Subroutine

Purpose

Retrieves an attribute string for a printer formatter.

Library

None (linked with the **pioformat** formatter driver)

Syntax

#include <piostruct.h>
piogetstr (*attrname***,** *bufrptr***,** *bufsiz***,** NULL**)**
char * *attrname***,*** *bufptr***;**
int *bufsiz***;**

Description

The **piogetstr** subroutine retrieves the specified attribute string from the Printer Attribute database and returns the string to the caller. In the course of retrieval, this subroutine also resolves any logic and any embedded references to other attribute strings or integers.

Parameters

attrname	Points to a two-character attribute name for a string. The attribute name must be defined in the database. It may optionally have been defined to the **piogetvals** subroutine as a variable string. The attribute should not be one that has been defined to the **piogetvals** subroutine as an integer.
bufptr	Points to where the constructed attribute string is to be stored.
bufsiz	Specifies the amount of memory that is available for storage of the string.

Note: The fourth parameter is reserved for future use. This parameter should be a NULL pointer.

Return Values

Upon successful completion, the **piogetstr** subroutine returns the length of the constructed string. The null character placed at the end of a constructed string by the **piogetstr** subroutine is not included in the length.

If the **piogetstr** subroutine detects an error, it issues an error message and terminates the print job.

Related Information

The **piogetvals** subroutine.

Understanding Embedded References in Printer Attribute Strings in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Adding a New Printer Type to Your System in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

---

piogetvals Subroutine

Purpose

Initializes a copy of Printer Attribute database variables for a printer formatter.

Library

None (linked with the **pioformat** formatter driver)

Syntax

```
#include <piostruct.h>
int piogetvals ( attrtable, NULL)
struct attrparms attrtable [];
```

Description

The **piogetvals** subroutine provides a way for a printer formatter's **setup** routine to define a list of printer attribute variables (and their characteristics) to the formatter driver. This routine, which is supplied by the formatter driver, allocates storage for the requested variables and uses the Printer Attribute database colon file to arrive at initial values.

The variables defined by the **piogetvals** subroutine are copies of variables in the database; they are used to hold current values of the variables. After the **piogetvals** subroutine returns pointers to each of the variables, the characteristics and memory location of each variable is known to both the formatter and the formatter driver. Subsequent changes to printer attribute values (made by the formatter while formatting an input data stream) are made to the newly defined variables, not to the database values. As a result of this scheme, the formatter driver always has access to the current value of each variable, but does not itself ever modify them.

The caller requests variables by filling in entries (an attribute name, its data type, and other characteristics) in the table pointed to by the *attrtable* parameter. For each entry, the **piogetvals** subroutine retrieves the requested attribute string in the Printer Attribute database and converts it, if necessary, into an actual value. The **piogetvals** subroutine then allocates memory for each of the variables, places the initial values there, and stores information about the variable (its name, data type, and memory location) in storage accessible to the **piogetopt**, **piocmdout**, and **piogetstr** subroutines.

Printer Attribute Variables

A Printer Attribute database is a colon file containing printer attribute values, which can be overridden at the time a print job is requested. These attributes can be constants or may be expressions with unresolved references to other attributes in them. These references are resolved before a database attribute is used to fill in the value of a requested variable.

Database attribute values, which are stored in the database as ASCII strings, have possible data types of string constant (the default), integer variable, or string variable. The requested variables should be either integers or strings. String variables are used primarily for strings that the formatter may need to modify during its processing. NULL characters have no special significance and are permissible within variable strings.

Data types for the requested variables are specified in the array of the **attrparms** structures pointed to by the *attrtable* parameter and are not specified at all in the Printer Attribute database. This means that for database values used exclusively by the formatter, only the formatter knows the actual data type of each value. The formatter uses the **piogetvals** routine in part to inform the formatter driver of the actual data type for database values that are not the default data type.

Converting a Database Attribute String to an Actual Value

Converting a database attribute string to an actual value involves two aspects. First, the **piogetvals** routine resolves any logic and any embedded references to other attribute strings, which yields a resolved string variable. Secondly, the data type of the requested variable must be checked. If this data type specifies a character string, then the resolved string is the final value, and it is stored in the memory allocated for it.

However, if the specified data type is integer variable, then the resolved string is converted to an integer. In this case, the *attrtable* entry for the attribute string is checked to determine how this conversion is to be performed. Either use the **atoi** subroutine for this purpose, or provide a pointer to a lookup table. After being converted to an integer, the value is stored in the memory allocated for it.

Using the **piogetvals** subroutine to convert database strings to integers as specified by the *attrtable* entries provides a table-driven procedure for the conversions. It also informs the formatter driver which values are integers and how strings that represent the integers can be converted into integer values. The

piogetopt, **piocmdout**, and **piogetstr** subroutines assume that the formatter has used the **piogetvals** subroutine to provide this information about the variables to the formatter driver.

When a formatter subsequently calls either the **piocmdout** subroutine or the **piogetstr** subroutine to access a string from the database, a global list of variables defined by the **piogetvals** subroutine is checked by the subroutine to see if the desired string has been defined. If so, then the value of the variable is taken from the memory location specified in the global list. If not, then the Printer Attribute database is consulted for the correct attribute string. Either the **piocmdout** or **piogetstr** subroutine scans the string to resolve any logic and any references to other strings or integers. The characteristics and memory locations of the variables, as remembered by the **piogetvals** subroutine, are used to obtain the current values of the variables.

Parameters

attrtable Points to a table of variables and their characteristics. The table is an array of **attrparms** structures, as defined in the **piostruct.h** file.

Note: The second parameter is reserved for future use. This parameter should be a NULL pointer.

Return Values

A return value of 0 indicates a successful operation. If the **piogetvals** subroutine detects an error, it issues an error message and terminates the print job.

Related Information

The **atoi** subroutine, **piocmdout** subroutine, **piogetopt** subroutine, **piogetstr** subroutine, the **setup** subroutine.

Understanding Embedded References in Printer Attribute Strings in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Adding a New Printer Type to Your System in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

---

piomsgout Subroutine

Purpose

Sends a message from a printer formatter.

Library

None (linked with the **pioformat** formatter driver)

Syntax

```
void piomsgout ( msgstr)
char *msgstr;
```

Description

The **piomsgout** subroutine should be used by printer formatters to send a message to the print job submitter, usually when an error is detected. This subroutine is supplied by the formatter driver.

If the formatter is running under the spooler, the message is displayed on the submitter's terminal if the submitter is logged on. Otherwise, the message is mailed to the submitter. If the formatter is not running under the spooler, the message is sent as standard error output.

The **piomsgout** subroutine has no return values.

Parameters

msgstr Points to the string of message text to be sent.

Related Information

Understanding Embedded References in Printer Attribute Strings in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Adding a New Printer Type to Your System in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

---

pioputattrs Subroutine

Purpose

Updates printer attribute values in a printer attribute database.

Library

libqb.a

Syntax

#include <piostruct.h>
int pioputattrs (*QueueName***,** *QueueDeviceName***,** *NumAttrElems***,** *AttrElemTable***)**
const char * *QueueName***, *** *QueueDeviceName***;**
unsigned short *NumAttrElems***;**
struct pioattr * *AttrElemTable***;**

Description

The **pioputattrs** subroutine can update with one call any number of attributes defined in a printer attribute database.

The combination of the *QueueName* and *QueueDeviceName* parameters identify a specific printer attribute database. The *QueueName* and *QueueDeviceName* parameters must be unique for a particular host.

Parameters

QueueName	Specifies the print-queue name. The print queue does not have to exist.
QueueDeviceName	Specifies the queue device name for the print queue name specified by the *QueueName* parameter. The queue device does not have to exist.
NumAttrElems	Specifies the number of attribute elements in the table specified by the *AttrElemTable* parameter.

AttrElemTable Points to a table of attribute element structures. Each structure element in the table specifies an attribute name, the type of value to be updated for the attribute, the value and length of the value, and a field for the return code of the update operation. The type of the value to be updated should be **PA_AVALT**. If a specified attribute is not valid, the specified value is put in the database. The format of each structure element is defined by the **pioattr** structure definition in the **/usr/include/piostruct.h** file.

Return Values

NumAttrElems Specifies the number of attribute elements for which the **pioputattrs** subroutine has successfully updated the specified values in the database.

-1 Indicates that an error occurred.

Examples

```
/* Array of elements to be passed to
pioputattrs() */
#define ATTR_ARRAY_NO (sizeof(attr_table)/sizeof(attr_table[0]))

struct pioattr attr_table[] = {
        {"_b", PA_AVALT, "2", 1, 0}, /* attribute record for   */
                                     /* _b (bottom margin)     */
        {"_i", PA_AVALT, "0", 1, 0}, /* attribute record for   */
                                     /* _i (left indentation)  */
        {"_t", PA_AVALT, "3", 1, 0}, /* attribute record for   */
                                     /* _t (top margin)        */
        {"sA", PA_AVALT, "CP851", 5, 0} /* attribute record    */
                                     /*for eS (country code)*/
}

...
const char                      *qnm = "ps";
const char                      *qdnm = "lp0";
int                             retno;
register const pioattr_t        *pap;

...
if((retno = pioputattrs(qnm,qdnm,ATTR_ARRAY_NO,attr_table)) ==-1)
        {(void) fprintf(stderr,"Fatal error in pioputattrs()\n");
...
}
```

pioputstatus Subroutine

Purpose

Puts job-status information for the specified print job into the specified status file.

Library

libqb.a

Syntax

```
#include <IN/stfile.h>

int pioputstatus(StatusFileDescriptor, VersionMagicNumber, StatusInformation)
int StatusFileDescriptor,  VersionMagicNumber;
const void * StatusInformation;
```

Description

The **pioputstatus** subroutine stores status information for a current print job.

The **pioputstatus** subroutine accepts a status structure containing print job information. This information includes queue name, queue device name, job number, and job status. The **pioputstatus** subroutine then stores the specified information in the specified status file.

The format of the status structure is determined by the version magic number specified by the *VersionMagicNumber* parameter. Each time there is a change in the status file structure for a new release, a unique number is assigned to the release's version magic number. This supports structure formats of previous releases.

Parameters

StatusFileDescriptor	Specifies the file descriptor of the status file. The *StatusFileDescriptor* parameter must specify a value of 3, because the print spooler always opens a status file with a file descriptor value of 3.
VersionMagicNumber	Specifies the version magic number that identifies the format of the status structure in which information is specified.
StatusInformation	Specifies a generic pointer to a status structure that contains print job status information that is to be stored in the status file.

Return Values

1 Indicates that the **pioputstatus** subroutine was successful.
-1 Indicates that an error occurred.

Subroutines for Writing a Print Formatter

The **pioformat** formatter driver requires a print formatter to contain the following function routines:

initialize	Performs printer initialization.
lineout	Formats a print line.
passthru	Passes through the input data stream without modification or formats the input data stream without assistance from the formatter driver.
restore	Restores the printer to its default state.
setup	Performs setup processing for the print formatter.

---

passthru Subroutine

Purpose

Passes through the input data stream without modification or formats the input data stream without assistance from the formatter driver.

Library

None (provided by the formatter).

Syntax

```
#include <piostruct.h>
int passthru ()
```

Description

The **passthru** subroutine is invoked by the formatter driver only if the **setup** subroutine returned a null pointer. If this is the case, the **passthru** subroutine is invoked (instead of the **lineout** subroutine) for one of the following reasons:

- The input data stream is to be passed through without modification.
- Formatting is done without the help of the formatter driver to handle vertical spacing.

Even if the data is being passed through from input to output without modification, a formatter program is used to initialize the printer before printing the file and to restore it to a known state afterward. However, gathering accounting information for an unknown data stream being passed through is difficult, if not impossible.

The **passthru** subroutine can also be used to format the input data stream if no help from the formatter driver for vertical spacing is needed. For example, if the only formatting to be done is to add a carrier-return control character to each linefeed control character, the **passthru** subroutine provides this simple task. The **passthru** subroutine can also count line feeds and form feeds to keep track of the page count. These counts can then be reported to the **log_pages** status subroutine, which is provided by the spooler.

Return Values

A return value of 0 indicates a successful operation. If the **passthru** subroutine detects an error, it uses the **piomsgout** subroutine to issue an error message. It then invokes the **pioexit** subroutine with a value of **PIOEXITBAD**. Note that if the **passthru** subroutine calls the **piocmdout** subroutine or the **piogetstr** subroutine and either of these detects an error, then the subroutine that detects the error automatically issues its own error message and terminates the print job.

Related Information

The **lineout** subroutine, **piocmdout** subroutine, **pioexit** subroutine, **piogetstr** subroutine, **piomsgout** subroutine, **setup** subroutine.

Adding a New Printer Type to Your System in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

---

restore Subroutine

Purpose

Restores the printer to its default state.

Library

None (provided by the formatter)

Syntax

```
#include <piostruct.h>
int restore ()
```

Description

The **restore** subroutine is invoked by the formatter driver after either the **lineout** subroutine or the **passthru** subroutine has reported that printing has completed.

If the **-J** flag passed from the command line has a nonzero value (True), the **initialize** subroutine should use the **piocmdout** subroutine to send a command string to the printer to restore the printer to its default state. This default state is defined by the attribute values in the database. Any variables referenced by the command string should be values from the database that have not been overridden by values from the command line. This can be accomplished by placing a **%o** escape sequence at the beginning of the command string.

Return Values

A return value of 0 indicates a successful operation. If the **restore** subroutine detects an error, it uses the **piomsgout** subroutine to issue an error message. The **restore** subroutine then invokes the **pioexit** subroutine with a value of **PIOEXITBAD**. If either the **piocmdout** or **piogetstr** subroutines detect an error, then the subroutine that detects the error issues an error message and terminates the print job.

Related Information

The **initialize** subroutine, **lineout** subroutine, **passthru** subroutine, **piocmdout** subroutine, **pioexit** subroutine, **piogetstr** subroutine.

Understanding Embedded References in Printer Attribute Strings in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Adding a New Printer Type to Your System in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

---

setup Subroutine

Purpose

Performs setup processing for the print formatter.

Library

None (provided by the formatter).

Syntax

#include <piostruct.h>
struct shar_vars *setup (*argc***,** *argv***,** *passthru***)**
unsigned *argc***;**
char * *argv* **[ ];**
int *passthru***;**

Description

The **setup** subroutine performs the following tasks:

- Invokes the **piogetvals** subroutine to initialize the database variables that the formatter uses.
- Processes the command line flags using the **piogetopt** subroutine.
- Validates the input parameters from the database and the command line.

The **setup** subroutine should not send commands or data to the printer since the formatter driver performs additional error checking when the **setup** subroutine returns.

Parameters

argc	Specifies the number of formatting arguments from the command line (including the command name).
argv	Points to a list of pointers to the formatting arguments.
passthru	Indicates whether the input data stream should be formatted (the *passthru* value is 0) or passed through without modification (the *passthru* value is1). The value for this parameter is the argument value for the **-#** flag specified to the **pioformat** formatter driver. If the **-#** flag is not specified, the *passthru* value is 0.

Return Values

Upon successful completion, the **setup** subroutine returns one of the following pointers:

- A pointer to a **shar_vars** structure that contains pointers to initialized vertical spacing variables. These variables are shared with the formatter driver, which provides vertical page movement.
- A null pointer, which indicates that the formatter handles its own vertical page movement or that the input data stream is to be passed through without modification. Vertical page movement includes top and bottom margins, new pages, initial pages to be skipped, and progress reports to the **qdaemon** daemon.

Returning a pointer to a **shar_vars** structure causes the formatter driver to invoke the formatter's lineout function for each line to be printed. Returning a null pointer causes the formatter driver to invoke the formatter's *passthru* function once instead.

If the **setup** subroutine detects an error, it uses the **piomsgout** subroutine to issue an error message. The **setup** subroutine then invokes the **pioexit** subroutine with a value of **PIOEXITBAD**. Note that if the **piogetvals**, **piogetopt**, **piocmdout**, or **piogetstr** subroutine detects an error, it automatically issues its own error message and terminates the print job.

Related Information

The **piocmdout** subroutine, **pioexit** subroutine, **piogetopt** subroutine, **piogetstr** subroutine, **piogetvals** subroutine, **piomsgout** subroutine.

The **qdaemon** daemon.

Understanding Embedded References in Printer Attribute Strings in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Adding a New Printer Type to Your System in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Example of Print Formatter in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

Chapter 5. SCSI Subsystem

scdisk SCSI Device Driver

Purpose

Supports the small computer system interface (SCSI) fixed disk, CD-ROM (compact disk read only memory), and read/write optical (optical memory) devices.

Syntax

```
#include <sys/devinfo.h>
#include <sys/scsi.h>
#include <sys/scdisk.h>
#include <sys/pcm.h>
#include <sys/mpio.h>
```

Device-Dependent Subroutines

Typical fixed disk, CD-ROM, and read/write optical drive operations are implemented using the **open**, **close, read**, **write**, and **ioctl** subroutines. The scdisk device driver has additional support added for MPIO capable devices.

open and close Subroutines

The **open** subroutine applies a reservation policy based on the ODM **reserve_policy** attribute. In the past, the **open** subroutine always applied a SCSI2 reserve. The **open** and **close** subroutines support working with multiple paths to a device if the device is a MPIO capable device.

The **openx** subroutine is intended primarily for use by diagnostic commands and utilities. Appropriate authority is required for execution. If an attempt is made to run the **open** subroutine without the proper authority, the subroutine returns a value of -1 and sets the errno global variable to a value of **EPERM**.

The *ext* parameter passed to the **openx** subroutine selects the operation to be used for the target device. The **/usr/include/sys/scsi.h** file defines possible values for the *ext* parameter.

The *ext* parameter can contain any combination of the following flag values logically ORed together:

SC_DIAGNOSTIC Places the selected device in Diagnostic mode. This mode is singularly entrant; that is, only one process at a time can open it. When a device is in Diagnostic mode, SCSI operations are performed during **open** or **close** operations, and error logging is disabled. In Diagnostic mode, only the **close** and **ioctl** subroutine operations are accepted. All other device-supported subroutines return a value of -1 and set the errno global variable to a value of **EACCES**.

A device can be opened in Diagnostic mode only if the target device is not currently opened. If an attempt is made to open a device in Diagnostic mode and the target device is already open, the subroutine returns a value of -1 and sets the errno global variable to a value of **EACCES**.

SC_FORCED_OPEN Forces a bus device reset, regardless of whether another initiator has the device reserved. The SCSI bus device reset is sent to the device before the **open** sequence begins. In other respects, the **open** operation runs normally.

SC_RETAIN_RESERVATION Retains the reservation of the device after a **close** operation by not issuing the release. This flag prevents other initiators from using the device unless they break the host machine's reservation.

SC_NO_RESERVE Prevents the reservation of a device during an **openx** subroutine call to that device. This operation is provided so a device can be controlled by two processors that synchronize their activity by their own software means.

SC_SINGLE Places the selected device in Exclusive Access mode. Only one process at a time can open a device in Exclusive Access mode.

A device can be opened in Exclusive Access mode only if the device is not currently open. If an attempt is made to open a device in Exclusive Access mode and the device is already open, the subroutine returns a value of -1 and sets the errno global variable to a value of **EBUSY**. If the **SC_DIAGNOSTIC** flag is specified along with the **SC_SINGLE** flag, the device is placed in Diagnostic mode.

SCSI Options to the openx Subroutine in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts* gives more specific information on the **open** operations.

readx and writex Subroutines

The **readx** and **writex** subroutines provide additional parameters which affect the raw data transfer. These subroutines pass the *ext* parameter, which specifies request options. The options are constructed by logically ORing zero or more of the following values:

HWRELOC Indicates a request for hardware relocation (safe relocation only)
UNSAFEREL Indicates a request for unsafe hardware relocation
WRITEV Indicates a request for write verification

ioctl Subroutine

ioctl subroutine operations that are used for the **scdisk** device driver are specific to the following categories:

- Fixed disk and read/write optical devices only
- CD-ROM devices only
- Fixed disk, CD-ROM, and read/write optical devices

Fixed Disk and Read/Write Optical Devices: The following **ioctl** operation is available for fixed disk and read/write optical devices only:

DKIOWRSE Provides a means for issuing a **write** command to the device and obtaining the target-device sense data when an error occurs. If the **DKIOWRSE** operation returns a value of -1 and the **status_validity** field is set to a value of **sc_valid_sense**, valid sense data is returned. Otherwise, target sense data is omitted.

The **DKIOWRSE** operation is provided for diagnostic use. It allows the limited use of the target device while operating in an active system environment. The *arg* parameter to the **DKIOWRSE** operation contains the address of an **sc_rdwrt** structure. This structure is defined in the **/usr/include/sys/scsi.h** file.

The **devinfo** structure defines the maximum transfer size for a **write** operation. If an attempt is made to transfer more than the maximum, the subroutine returns a value of -1 and sets the errno global variable to a value of **EINVAL**. Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

CD-ROM Devices Only: The following **ioctl** operation is available for CD-ROM devices only:

CDIOCMD Allows SCSI commands to be issued directly to the attached CD-ROM device. The **CDIOCMD** operation preserves binary compatibility for CD-ROM applications that were compiled on earlier releases of the operating system. It is recommended that newly written CD-ROM applications use the **DKIOCMD** operation instead. For the **CDIOCMD** operation, the device must be opened in Diagnostic mode. The **CDIOCMD** operation parameter specifies the address of a **sc_iocmd** structure. This structure is defined in the **/usr/include/sys/scsi.h** file.

If this operation is attempted on a device other than CD-ROM, it is interpreted as a **DKIORDSE** operation. In this case, the *arg* parameter is treated as an **sc_rdwrt** structure.

If the **CDIOCMD** operation is attempted on a device not in Diagnostic mode, the subroutine returns a value of -1 and sets the errno global variable to a value of **EACCES**. Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.
Note: Diagnostic mode is required only for the **CDIOCMD** and **DKIOCMD** operations.

Fixed Disk, CD-ROM, and Read/Write Optical Devices: The following **ioctl** operations are available for fixed disk, CD-ROM, and read/write optical devices:

IOCINFO Returns the **devinfo** structure defined in the **/usr/include/sys/devinfo.h** file. The **IOCINFO** operation is the only operation defined for all device drivers that use the **ioctl** subroutine. The remaining operations discussed in this article are all specific to fixed disk, CD-ROM, and read/write optical devices.

DKIORDSE Provides a means for issuing a **read** command to the device and obtaining the target-device sense data when an error occurs. If the **DKIORDSE** operation returns a value of -1 and the **status_validity** field is set to a value of **sc_valid_sense**, valid sense data is returned. Otherwise, target sense data is omitted.

The **DKIORDSE** operation is provided for diagnostic use. It allows the limited use of the target device while operating in an active system environment. The *arg* parameter to the **DKIORDSE** operation contains the address of an **sc_rdwrt** structure. This structure is defined in the **/usr/include/sys/scsi.h** file.

The **devinfo** structure defines the maximum transfer size for a **read** operation. If an attempt is made to transfer more than the maximum, the subroutine returns a value of -1 and sets the errno global variable to a value of **EINVAL**. Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.
Note: The **CDIORDSE** operation may be substituted for the **DKIORDSE** operation when issuing a **read** command to and obtaining sense data from a CD-ROM device. **DKIORDSE** is the recommended operation.

DKIOCMD When the device has been successfully opened in the Diagnostic mode, the **DKIOCMD** operation provides the means for issuing any SCSI command to the specified device. If the **DKIOCMD** operation is issued when the device is not in Diagnostic mode, the subroutine returns a value of -1 and sets the errno global variable to a value of **EACCES**. The device driver performs no error recovery or logging on failures of this operation.

The SCSI status byte and the adapter status bytes are returned through the *arg* parameter, which contains the address of a **sc_iocmd** structure (defined in the **/usr/include/sys/scsi.h** file). If the **DKIOCMD** operation fails, the subroutine returns a value of -1 and sets the errno global variable to a nonzero value. In this case, the caller should evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions should be taken.

The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum, the subroutine returns a value of -1 and sets the errno global variable to a value of **EINVAL**. Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.
Note: Diagnostic mode is required only for the **CDIOCMD** and **DKIOCMD** operations.

DKPMR Issues a SCSI prevent media removal command when the device has been successfully opened. This command prevents media from being ejected until the device is closed, powered off and back on, or until a **DKAMR** operation is issued. The *arg* parameter for the **DKPMR** operation is null. If the **DKPMR** operation is successful, the subroutine returns a value of 0. If the device is a SCSI fixed disk, the **DKPMR** operation fails, and the subroutine returns a value of -1 and sets the errno global variable to a value of **EINVAL**. If the **DKPMR** operation fails for any other reason, the subroutine returns a value of -1 and sets the errno global variable to a value of **EIO**.

DKAMR Issues an allow media removal command when the device has been successfully opened. As a result media can be ejected using either the drive's eject button or the **DKEJECT** operation. The *arg* parameter for this ioctl is null. If the **DKAMR** operation is successful, the subroutine returns a value of 0. If the device is a SCSI fixed disk, the **DKAMR** operation fails, and the subroutine returns a value of -1 and sets the errno global variable to a value of **EINVAL**. For any other failure of this operation, the subroutine returns a value of -1 and sets the errno global variable to a value of **EIO**.

DKEJECT Issues an eject media command to the drive when the device has been successfully opened. The *arg* parameter for this operation is null. If the **DKEJECT** operation is successful, the subroutine returns a value of 0. If the device is a SCSI fixed disk, the **DKEJECT** operation fails, and the subroutine returns a value of -1 and sets the errno global variable to a value of **EINVAL**. For any other failure of this operation, the subroutine returns a value of -1 and sets the errno global variable to a value of **EIO**.

DKFORMAT Issues a format unit command to the specified device when the device has been successfully opened.

If the *arg* parameter for this operation is null, the format unit sets the format options valid (FOV) bit to 0 (that is, it uses the drive's default setting). If the *arg* parameter for the **DKFORMAT** operation is not null, the first byte of the defect list header is set to the value specified in the first byte addressed by the *arg* parameter. This allows the creation of applications to format a particular type of read/write optical media uniquely.

The driver initially tries to set the FmtData and CmpLst bits to 0. If that fails, the driver tries the remaining three permutations of these bits. If all four permutations fail, this operation fails, and the subroutine sets the errno global variable to a value of **EIO**.

If the **DKFORMAT** operation is specified for a fixed disk, the subroutine returns a value of -1 and sets the errno global variable to a value of **EINVAL**. If the **DKFORMAT** operation is attempted when the device is not in Exclusive Access mode, the subroutine returns a value of -1 and sets the errno global variable to a value of **EACCES**. If the media is write-protected, the subroutine returns a value of -1 and sets the errno global variable to a value of **EWRPROTECT**. If the format unit exceeds its timeout value, the subroutine returns a value of -1 and sets the errno global variable to a value of **ETIMEDOUT**. For any other failure of this operation, the subroutine returns a value of -1 and sets the errno global variable to a value of **EIO**.

DKAUDIO Issues play audio commands to the specified device and controls the volume on the device's output ports. Play audio commands include: play, pause, resume, stop, determine the number of tracks, and determine the status of a current audio operation. The **DKAUDIO** operation plays audio only through the CD-ROM drive's output ports. The *arg* parameter of this operation is the address of a **cd_audio_cmds** structure, which is defined in the **/usr/include/sys/scdisk.h** file. Exclusive Access mode is required.

If **DKAUDIO** operation is attempted when the device's audio-supported attribute is set to No, the subroutine returns a value of -1 and sets the errno global variable to a value of **EINVAL**. If the **DKAUDIO** operation fails, the subroutine returns a value of -1 and sets the errno global variable to a nonzero value. In this case, the caller should evaluate the returned status bytes to determine why the operation failed and what recovery actions should be taken.

DK_CD_MODE Determines or changes the CD-ROM data mode for the specified device. The CD-ROM data mode specifies what block size and special file are used for data read across the SCSI bus from the device. The **DK_CD_MODE** operation supports the following CD-ROM data modes:

CD-ROM Data Mode 1
512-byte block size through both raw (**dev/rcd***) and block special (**/dev/cd***) files

CD-ROM Data Mode 2 Form 1
2048-byte block size through both raw (**dev/rcd***) and block special (**/dev/cd***) files

CD-ROM Data Mode 2 Form 2
2336-byte block size through the raw (**dev/rcd***) special file only

CD-DA (Compact Disc Digital Audio)
2352-byte block size through the raw (**dev/rcd***) special file only

DVD-ROM
2048-byte block size through both raw (**/dev/rcd***) and block special (**/dev/cd***) files

DVD-RAM
2048-byte block size through both raw (**/dev/rcd***) and block special (**/dev/cd***) files

DVD-RW
2048-byte block size through both raw (**/dev/rcd***) and block special (**/dev/cd***) files

The **DK_CD_MODE** *arg* parameter contains the address of the **mode_form_op** structure defined in the **/usr/include/sys/scdisk.h** file. To have the **DK_CD_MODE** operation determine or change the CD-ROM data mode, set the `action` field of the **change_mode_form** structure to one of the following values:

CD_GET_MODE
Returns the current CD-ROM data mode in the `cd_mode_form` field of the **mode_form_op** structure, when the device has been successfully opened.

CD_CHG_MODE
Changes the CD-ROM data mode to the mode specified in the `cd_mode_form` field of the **mode_form_op** structure, when the device has been successfully opened in the Exclusive Access mode.

If a CD-ROM has not been configured for different data modes (via mode-select density codes), and an attempt is made to change the CD-ROM data mode (by setting the `action` field of the **change_mode_form** structure set to **CD_CHG_MODE**), the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. Attempts to change the CD-ROM mode to any of the DVD modes will also result in a return value of -1 and the **errno** global variable set to **EINVAL**.

If the **DK_CD_MODE** operation for **CD_CHG_MODE** is attempted when the device is not in Exclusive Access mode, the subroutine returns a value of -1 and sets the errno global variable to a value of **EACCES**. For any other failure of this operation, the subroutine returns a value of -1 and sets the errno global variable to a value of **EIO**.

DK_PASSTHRU When the device has been successfully opened, the **DK_PASSTHRU** operation provides the means for issuing any SCSI command to the specified device. The device driver will perform limited error recovery if this operation fails. The **DK_PASSTHRU** operation differs from the **DKIOCMD** operation in that it does not require an **openx** command with the *ext* argument of **SC_DIAGNOSTIC**. Because of this, a **DK_PASSTHRU** operation can be issued to devices that are in use by other operations.

The SCSI status byte and the adapter status bytes are returned through the *arg* parameter, which contains the address of a **sc_passthru** structure (defined in the **/usr/include/sys/scsi.h** file). If the **DK_PASSTHRU** operation fails, the subroutine returns a value of -1 and sets the errno global variable to a nonzero value. If this happens the caller should evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions should be taken.

If a **DK_PASSTHRU** operation fails because a field in the **sc_passthru** structure has an invalid value, the subroutine will return a value of -1 and set the errno global variable to **EINVAL**. The **einval_arg** field will be set to the field number (starting with 1 for the version field) of the field that had an invalid value. A value of 0 for the **einval_arg** field indicates no additional information on the failure is available.

DK_PASSTHRU (continued) **DK_PASSTHRU** operations are further subdivided into requests which quiesce other I/O prior to issuing the request and requests that do not quiesce I/O. These subdivisions are based on the **devflags** field of the **sc_passthru** structure. When the **devflags** field of the **sc_passthru** structure has a value of **SC_MIX_IO**, the **DK_PASSTHRU** operation will be mixed with other I/O requests. **SC_MIX_IO** requests that write data to devices are prohibited and will fail. When this happens -1 is returned, and the errno global variable is set to **EINVAL**. When the **devflags** field of the **sc_passthru** structure has a value of **SC_QUIESCE_IO**, all other I/O requests will be quiesced before the **DK_PASSTHRU** request is issued to the device. If an **SC_QUIESCE_IO** request has its **timeout_value** field set to 0, the **DK_PASSTHRU** request will be failed with a return code of -1, the errno global variable will be set to **EINVAL**, and the **einval_arg** field will be set to a value of **SC_PASSTHRU_INV_TO** (defined in the **/usr/include/sys/scsi.h** file). If an **SC_QUIESCE_IO** request has a nonzero timeout value that is too large for the device, the **DK_PASSTHRU** request will be failed with a return code of -1, the errno global variable will be set to **EINVAL**, the **einval_arg** field will be set to a value of **SC_PASSTHRU_INV_TO** (defined in the **/usr/include/sys/scsi.h** file), and the **timeout_value** will be set to the largest allowed value.

The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum transfer size, the subroutine returns a value of -1, sets the errno global variable to a value of **EINVAL**, and sets the **einval_arg** field to a value of **SC_PASSTHRU_INV_D_LEN** (defined in the **/usr/include/sys/scsi.h** file).

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

DK_RWBUFFER When the device has been successfully opened, the **DK_RWBUFFER** operation provides the means for issuing one or more SCSI Write Buffer commands to the specified device. The device driver will perform full error recovery upon failures of this operation. The **DK_RWBUFFER** operation differs from the **DKIOCMD** operation in that it does not require an exclusive open of the device (for example, **openx** with the *ext* argument of **SC_DIAGNOSTIC**). Thus, a **DK_RWBUFFER** operation can be issued to devices that are in use by others. It can be used in conjunction with the **DK_PASSTHRU** ioctl, which (like **DK_RWBUFFER**) does not require an exclusive open of the device.

The *arg* parameter contains the address of a **sc_rwbuffer** structure (defined in the **/usr/include/sys/scsi.h** file). Before the **DK_RWBUFFER** ioctl is invoked, the fields of this structure should be set according to the desired behavior. The **mode** field corresponds to the **mode** field of the SCSI Command Descriptor Block (CDB) as defined in the *SCSI Primary Commands (SPC) Specification*. Supported modes are listed in the header file **/usr/include/sys/scsi.h**.

The device driver will quiesce all other I/O from the initiator issuing the Write Buffer ioctl until the entire operation completes. Once the Write Buffer ioctl completes, all quiesced I/O will be resumed.

The SCSI status byte and the adapter status bytes are returned through the *arg* parameter, which contains the address of a **sc_rwbuffer** structure (defined in the **/usr/include/sys/scsi.h** file). If the **DK_RWBUFFER** operation fails, the subroutine returns a value of -1 and sets the **errno** global variable to a nonzero value. In this case, the caller should evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions should be taken.

If a **DK_RWBUFFER** operation fails because a field in the **sc_rwbuffer** structure has an invalid value, the subroutine will return a value of -1 and set the **errno** global variable to **EINVAL**.

The **DK_RWBUFFER** ioctl allows the user to issue multiple SCSI Write Buffer commands (CDBs) to the device through a single ioctl invocation. This is useful for applications such as microcode download where the user provides a pointer to the entire microcode image, but, due to size restrictions of the device buffer(s), desires that the images be sent in fragments until the entire download is complete.

If the **DK_RWBUFFER** ioctl is invoked with the **fragment_size** member of the **sc_rwbuffer** struct equal to **data_length**, a single Write Buffer command will be issued to the device with the **buffer_offset** and **buffer_ID** of the SCSI CDB set to the values provided in the **sc_rwbuffer** struct.

DK_RWBUFFER (continued) If **data_length** is greater than **fragment_size** and **fragment_size** is a nonzero value, multiple Write Buffer commands will be issued to the device. The number of Write Buffer commands (SCSI CDBs) issued will be calculated by dividing the **data_length** by the desired **fragment_size**. This value will be incremented by 1 if the **data_length** is not an even multiple of **fragment_size**, and the final data transfer will be the size of this residual amount. For each Write Buffer command issued, the **buffer_offset** will be set to the value provided in the **sc_rwbuffer** struct (microcode downloads to SCSD devices requires this to be set to 0). For the first command issued, the **buffer_ID** will be set to the value provided in the **sc_rwbuffer** struct. For each subsequent Write Buffer command issued, the **buffer_ID** will be incremented by 1 until all fragments have been sent. Writing to noncontiguous **buffer_ID**s through a single **DK_RWBUFFER** ioctl is not supported. If this functionality is desired, multiple **DK_RWBUFFER** ioctls must be issued with the **buffer_ID** set appropriately for each invocation.
Note: No I/O is quiesced between ioctl invocations.

If **fragment_size** is set to zero, an **errno** of **EINVAL** will be returned. If the desire is to send the entire buffer with one SCSI Write buffer command, this field should be set equal to **data_length**. An error of **EINVAL** will also be returned if the **fragment_size** is greater than the **data_length**.

The Parameter List Length (**fragment_size**) plus the Buffer Offset can not exceed the capacity of the specified buffer of the device. It is the responsibility of the caller of the Write Buffer ioctl to ensure that the **fragment_size** setting satisfies this requirement. A **fragment_size** larger than the device can accommodate will result in a SCSI error at the device, and the Write Buffer ioctl will simply report this error but take no action to recover.

The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum transfer size, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request sense data for a particular device.

DKPATHIOCMD This command is only available for MPIO capable devices. The **DKPATHIOCMD** command takes as input a pointer argument which points to a single **scdisk_pathiocmd** structure. The **DKPATHIOCMD** command behaves exactly like the **DKIOCMD** command, except that the input path is used rather than normal path selection. The **DKPATHIOCMD** path is used for the **DKIOCMD** command regardless of any path specified by a **DKPATHFORCE** ioctl command. A path cannot be unconfigured while it is being forced.

DKPATHFORCE This command is only available for MPIO capable devices. The **DKPATHFORCE** command takes as input a ushort path id. The path id should correspond to one of the path ids in CuPath ODM. The path id specifies a path to be used for all subsequent I/O commands, overriding any previous **DKPATHFORCE** path. A zero argument specifies that path forcing is terminated and that normal MPIO path selection is to be resumed. I/O commands sent in with the **DKPATHIOCMD** command will override the **DKPATHFORCE** option and send the I/O down the path specified in **scdisk_pathiocmd** structure.

DKPATHRWBUFFER This command is only available for MPIO capable devices. The **DKPATHRWBUFFER** command takes as input a pointer argument which points to a single **scdisk_pathiocmd** structure. The **DKPATHRWBUFFER** command behaves exactly like the **DKRWBUFFER** command, except that the input path is used rather than normal path selection. The **DKPATHRWBUFFER** path is used for the **DKRWBUFFER** command regardless of any path specified by a **DKPATHFORCE** ioctl command.

DKPATHPASSTHRU This command is only available for MPIO capable devices. The **DKPATHPASSTHRU** command takes as input a pointer argument which points to a single **scdisk_pathiocmd** structure. The **DKPATHPASSTHRU** command behaves exactly like the **DKPASSTHRU** command, except that the input path is used rather than normal path selection. The **DKPATHPASSTHRU** path is used for the **DKPASSTHRU** command regardless of any path specified by a **DKPATHFORCE** ioctl command.

DKPCMPASSTHRU This command is only available for MPIO capable devices. The **DKPCMPASSTHRU** command takes as input a structure which is PCM specific, it is not defined by AIX. The PCM specific structure is passed to the PCM directly. This structure can be used to move information to or from a PCM.

Device Requirements

SCSI fixed disk, CD-ROM, and read/write optical drives have the following hardware requirements:

- SCSI fixed disks and read/write optical drives must support a block size of 512 bytes per block.
- If mode sense is supported, the write-protection (WP) bit must also be supported for SCSI fixed disks and read/write optical drives.
- SCSI fixed disks and read/write optical drives must report the hardware retry count in bytes 16 and 17 of the request sense data for recovered errors. If the fixed disk or read/write optical drive does not support this, the system error log may indicate premature drive failure.
- SCSI CD-ROM and read/write optical drives must support the 10-byte SCSI read command.
- SCSI fixed disks and read/write optical drives must support the SCSI write and verify command and the 6-byte SCSI write command.
- To use the **format** command operation on read/write optical media, the drive must support setting the format options valid (FOV) bit to 0 for the defect list header of the SCSI format unit command. If the drive does not support this, the user can write an application for the drive so that it formats media using the **DKFORMAT** operation.
- If a SCSI CD-ROM drive uses **CD_ROM Data Mode 1**, it must support a block size of 512 bytes per block.
- If a SCSI CD-ROM drive uses **CD_ROM data Mode 2 Form 1**, it must support a block size of 2048 bytes per block.
- If a SCSI CD-ROM drive uses **CD_ROM data Mode 2 Form 2**, it must support a block size of 2336 bytes per block.
- If a SCSI CD-ROM drive uses **CD_DA** mode, it must support a block size of 2352 bytes per block.
- To control volume using the **DKAUDIO** (play audio) operation, the device must support SCSI-2 mode data page 0xE.
- To use the **DKAUDIO** (play audio) operation, the device must support the following SCSI-2 optional commands:
  - read sub-channel
  - pause resume
  - play audio MSF
  - play audio track index
  - read TOC

Error Conditions

Possible **errno** values for **ioctl**, **open**, **read**, and **write** subroutines when using the **scdisk** device driver include:

EACCES Indicates one of the following circumstances:
- An attempt was made to open a device currently open in Diagnostic or Exclusive Access mode.
- An attempt was made to open a Diagnostic mode session on a device already open.
- The user attempted a subroutine other than an **ioctl** or **close** subroutine while in Diagnostic mode.
- A **DKIOCMD** or **CDIOCMD** operation was attempted on a device not in Diagnostic mode.
- **A DK_CD_MODE ioctl** subroutine operation was attempted on a device not in Exclusive Access mode.
- A **DKFORMAT** operation was attempted on a device not in Exclusive Access mode.

EBUSY Indicates one of the following circumstances:
- An attempt was made to open a session in Exclusive Access mode on a device already opened.
- The target device is reserved by another initiator.

EFAULT Indicates an illegal user address.

EFORMAT Indicates the target device has unformatted media or media in an incompatible format.

EINPROGRESS Indicates a CD-ROM drive has a play-audio operation in progress.

EINVAL Indicates one of the following circumstances:

- A **DKAUDIO** (play-audio) operation was attempted for a device that is not configured to use the SCSI-2 play-audio commands.
- The **read** or **write** subroutine supplied an *nbyte* parameter that is not an even multiple of the block size.
- A sense data buffer length of greater than 255 bytes is not valid for a **CDIORDSE**, **DKIOWRSE**, or **DKIORDSE ioctl** subroutine operation.
- The data buffer length exceeded the maximum defined in the **devinfo** structure for a **CDIORDSE**, **CDIOCMD**, **DKIORDSE**, **DKIOWRSE**, or **DKIOCMD ioctl** subroutine operation.
- An unsupported **ioctl** subroutine operation was attempted.
- A data buffer length greater than that allowed by the CD-ROM drive is not valid for a **CDIOCMD ioctl** subroutine operation.
- An attempt was made to configure a device that is still open.
- An illegal configuration command has been given.
- A **DKPMR** (Prevent Media Removal), **DKAMR** (Allow Media Removal), or **DKEJECT** (Eject Media) command was sent to a device that does not support removable media.
- A **DKEJECT** (Eject Media) command was sent to a device that currently has its media locked in the drive.
- The data buffer length exceeded the maximum defined for a **strategy** operation.

EIO Indicates one of the following circumstances:

- The target device cannot be located or is not responding.
- The target device has indicated an unrecoverable hardware error.

EMEDIA Indicates one of the following circumstances:

- The target device has indicated an unrecoverable media error.
- The media was changed.

EMFILE Indicates an **open** operation was attempted for an adapter that already has the maximum permissible number of opened devices.

ENODEV Indicates one of the following circumstances:

- An attempt was made to access an undefined device.
- An attempt was made to close an undefined device.

ENOTREADY Indicates no media is in the drive.

ENXIO Indicates one of the following circumstances:

- The **ioctl** subroutine supplied an invalid parameter.
- A **read** or **write** operation was attempted beyond the end of the fixed disk.

EPERM Indicates the attempted subroutine requires appropriate authority.

ESTALE Indicates a read-only optical disk was ejected (without first being closed by the user) and then either reinserted or replaced with a second optical disk.

ETIMEDOUT Indicates an I/O operation has exceeded the given timer value.

EWRPROTECT Indicates one of the following circumstances:

- An **open** operation requesting **read/write** mode was attempted on read-only media.
- A **write** operation was attempted to read-only media.

Reliability and Serviceability Information

SCSI fixed disk devices, CD-ROM drives, and read/write optical drives return the following errors:

ABORTED COMMAND Indicates the device ended the command

ADAPTER ERRORS Indicates the adapter returned an error

GOOD COMPLETION	Indicates the command completed successfully
HARDWARE ERROR	Indicates an unrecoverable hardware failure occurred during command execution or during a self-test
ILLEGAL REQUEST	Indicates an illegal command or command parameter
MEDIUM ERROR	Indicates the command ended with an unrecoverable media error condition
NOT READY	Indicates the logical unit is offline or media is missing
RECOVERED ERROR	Indicates the command was successful after some recovery was applied
UNIT ATTENTION	Indicates the device has been reset or the power has been turned on

Error Record Values for Media Errors

The fields defined in the error record template for fixed disk, CD-ROM, and read/write optical media errors are:

`Comment`	Indicates fixed disk, CD-ROM, or read/write optical media error.
`Class`	Equals a value of H, which indicates a hardware error.
`Report`	Equals a value of True, which indicates this error should be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error log entry should be created when this error occurs.
`Alert`	Equals a value of False, which indicates this error is not alertable.
`Err_Type`	Equals a value of Perm, which indicates a permanent failure.
`Err_Desc`	Equals a value of 1312, which indicates a disk operation failure.
`Prob_Causes`	Equals a value of 5000, which indicates media.
`User_Causes`	Equals a value of 5100, which indicates the media is defective.
`User_Actions`	Equals the following values: • 0000, which indicates problem-determination procedures should be performed • 1601, which indicates the removable media should be replaced and retried
`Inst_Causes`	None.
`Inst_Actions`	None.
`Fail_Causes`	Equals the following values: • 5000, which indicates a media failure • 6310, which indicates a disk drive failure
`Fail_Actions`	Equals the following values: • 0000, which indicates problem-determination procedures should be performed • 1601, which indicates the removable media should be replaced and retried
`Detail_Data`	Equals a value of 156, 11, HEX. This value indicates hexadecimal format.

Note: The `Detail_Data` field in the **err_rec** structure contains the **sc_error_log_df** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **sc_error_log_df** structure is defined in the **/usr/include/sys/scsi.h** file.

The **sc_error_log_df** structure contains the following fields:

`req_sense_data`
Contains the request-sense information from the particular device that had the error, if it is valid.

`reserved2`
Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.

`reserved3`
Contains the number of bytes read since the segment count was last increased.

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Error Record Values for Hardware Errors

The fields defined in the error record template for fixed disk, CD-ROM, and read/write optical hardware errors, as well as hard-aborted command errors are:

`Comment` Indicates fixed disk, CD-ROM, or read/write optical hardware error.
`Class` Equals a value of H, which indicates a hardware error.
`Report` Equals a value of True, which indicates this error should be included when an error report is generated.
`Log` Equals a value of True, which indicates an error log entry should be created when this error occurs.
`Alert` Equal to a value of FALSE, which indicates this error is not alertable.
`Err_Type` Equals a value of Perm, which indicates a permanent failure.
`Err_Desc` Equals a value of 1312, which indicates a disk operation failure.
`Prob_Causes` Equals a value of 6310, which indicates disk drive.
`User_Causes` None.
`User_Actions` None.
`Inst_Causes` None.
`Inst_Actions` None.
`Fail_Causes` Equals the following values:

- 6310, which indicates a disk drive failure
- 6330, which indicates a disk drive electronics failure

`Fail_Actions` Equals a value of 0000, which indicates problem-determination procedures should be performed.
`Detail_Data` Equals a value of 156, 11, HEX. This value indicates hexadecimal format.

Note: The `Detail_Data` field in the **err_rec** structure contains the **sc_error_log_df** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **sc_error_log_df** structure is defined in the **/usr/include/sys/scsi.h** file.

The **sc_error_log_df** structure contains the following fields:

req_sense_data
Contains the request-sense information from the particular device that had the error, if it is valid.

reserved2
Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.

reserved3
Contains the number of bytes read since the segment count was last increased.

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Error Record Values for Adapter-Detected Hardware Failures

The fields defined in the error record template for fixed disk, CD-ROM, and read/write optical media errors adapter-detected hardware errors are:

`Comment` Indicates adapter-detected fixed disk, CD-ROM, or read/write optical hardware failure.
`Class` Equals a value of H, which indicates a hardware error.
`Report` Equals a value of True, which indicates this error should be included when an error report is generated.
`Log` Equals a value of True, which indicates an error-log entry should be created when this error occurs.
`Alert` Equal to a value of FALSE, which indicates this error is not alertable.
`Err_Type` Equals a value of Perm, which indicates a permanent failure.
`Err_Desc` Equals a value of 1312, which indicates a disk operation failure.

Prob_Causes Equals the following values:

- 3452, which indicates a device cable failure
- 6310, which indicates a disk drive failure

User_Causes None.
User_Actions None.
Inst_Causes None.
Inst_Actions None.
Fail_Causes Equals the following values:

- 3452, which indicates a storage device cable failure
- 6310, which indicates a disk drive failure
- 6330, which indicates a disk-drive electronics failure

Fail_Actions Equals a value of 0000, which indicates problem-determination procedures should be performed.
Detail_Data Equals a value of 156, 11, HEX. This value indicates hexadecimal format.

Note: The Detail_Data field in the **err_rec** structure contains the **sc_error_log_df** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **sc_error_log_df** structure is defined in the **/usr/include/sys/scsi.h** file.

The **sc_error_log_df** structure contains the following fields:

req_sense_data
Contains the request-sense information from the particular device that had the error, if it is valid.

reserved2
Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.

reserved3
Contains the number of bytes read since the segment count was last increased.

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Error Record Values for Recovered Errors

The fields defined in the error record template for fixed disk, CD-ROM, and read/write optical media errors recovered errors are:

Comment Indicates fixed disk, CD-ROM, or read/write optical recovered error.
Class Equals a value of H, which indicates a hardware error.
Report Equals a value of True, which indicates this error should be included when an error report is generated.
Log Equals a value of True, which indicates an error log entry should be created when this error occurs.
Alert Equal to a value of FALSE, which indicates this error is not alertable.
Err_Type Equals a value of Temp, which indicates a temporary failure.
Err_Desc Equals a value of 1312, which indicates a physical volume operation failure.
Prob_Causes Equals the following values:

- 5000, which indicates a media failure
- 6310, which indicates a disk drive failure

User_Causes Equals a value of 5100, which indicates media is defective.
User_Actions Equals the following values:

- 0000, which indicates problem-determination procedures should be performed
- 1601, which indicates the removable media should be replaced and retried

Inst_Causes None.
Inst_Actions None.

`Fail_Causes` Equals the following values:

- 5000, which indicates a media failure
- 6310, which indicates a disk drive failure

`Fail_Actions` Equals the following values:

- 0000, which indicates problem-determination procedures should be performed
- 1601, which indicates the removable media should be replaced and retried

`Detail_Data` Equals a value of 156, 11, HEX. This value indicates hexadecimal format.

Note: The `Detail_Data` field in the **err_rec** structure contains the **sc_error_log_df** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **sc_error_log_df** structure is defined in the **/usr/include/sys/scsi.h** file.

The **sc_error_log_df** structure contains the following fields:

req_sense_data
Contains the request-sense information from the particular device that had the error, if it is valid.

reserved2
Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.

reserved3
Contains the number of bytes read since the segment count was last increased.

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Error Record Values for Unknown Errors

The fields defined in the error record template for fixed disk, CD-ROM, and read/write optical media errors unknown errors are:

`Comment` Indicates fixed disk, CD-ROM, or read/write optical unknown failure.
`Class` Equals a value of H, which indicates a hardware error.
`Report` Equals a value of True, which indicates this error should be included when an error report is generated.
`Log` Equals a value of True, which indicates an error log entry should be created when this error occurs.
`Alert` Equal to a value of FALSE, which indicates this error is not alertable.
`Err_Type` Equals a value of Unkn, which indicates the type of error is unknown.
`Err_Desc` Equals a value of FE00, which indicates an undetermined error.
`Prob_Causes` Equals the following values:

- 3300, which indicates an adapter failure
- 5000, which indicates a media failure
- 6310, which indicates a disk drive failure

`User_Causes` None.
`User_Actions` None.
`Inst_Causes` None.
`Inst_Actions` None.
`Fail_Causes` Equals a value of FFFF, which indicates the failure causes are unknown.
`Fail_Actions` Equals the following values:

- 0000, which indicates problem-determination procedures should be performed
- 1601, which indicates the removable media should be replaced and retried

`Detail_Data` Equals a value of 156, 11, HEX. This value indicates hexadecimal format.

Note: The `Detail_Data` field in the **err_rec** structure contains the **sc_error_log_df** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **sc_error_log_df** structure is defined in the **/usr/include/sys/scsi.h** file.

The **sc_error_log_df** structure contains the following fields:

req_sense_data
Contains the request-sense information from the particular device that had the error, if it is valid.

reserved2
Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.

reserved3
Contains the number of bytes read since the segment count was last increased.

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Special Files

The **scdisk** SCSI device driver uses raw and block special files in performing its functions.

Attention: Data corruption, loss of data, or loss of system integrity (system crash) will occur if devices supporting paging, logical volumes, or mounted file systems are accessed using block special files. Block special files are provided for logical volumes and disk devices and are solely for system use in managing file systems, paging devices, and logical volumes. These files should not be used for other purposes.

The special files used by the **scdisk** device driver include the following (listed by type of device):

- Fixed disk devices:

/dev/rhdisk0, **/dev/rhdisk1**,..., **/dev/rhdisk***n*	Provides an interface to allow SCSI device drivers character access (raw I/O access and control functions) to SCSI fixed disks.
/dev/hdisk0, **/dev/hdisk1**,..., **/dev/hdisk***n*	Provides an interface to allow SCSI device drivers block I/O access to SCSI fixed disks.

- CD-ROM devices:

/dev/rcd0, **/dev/rcd1**,..., **/dev/rcd***n*	Provides an interface to allow SCSI device drivers character access(raw I/O access and control functions) to SCSI CD-ROM disks.
/dev/cd0, **/dev/cd1**,..., **/dev/cd***n*	Provides an interface to allow SCSI device drivers block I/O access to SCSI CD-ROM disks.

- Read/write optical devices:

/dev/romd0, **/dev/romd1**,..., **/dev/romd***n*	Provides an interface to allow SCSI device drivers character access (raw I/O access and control functions) to SCSI read/write optical devices.
/dev/omd0, **/dev/omd1**,..., **/dev/omd***n*	Provides an interface to allow SCSI device drivers block I/O access to SCSI read/write optical devices.

Note: The prefix **r** on a special file name indicates the drive is accessed as a raw device rather than a block device. Performing raw I/O with a fixed disk, CD-ROM, or read/write optical drive requires

that all data transfers be in multiples of the device block size. All **lseek** subroutines that are made to the raw device driver must result in a file pointer value that is a multiple of the device block size.

Related Information

Special Files Overview in *AIX 5L Version 5.2 Files Reference*.

SCSI Subsystem Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

A Typical Initiator-Mode SCSI Driver Transaction Sequence in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Required SCSI Adapter Device Driver ioctl Commands in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding the Execution of Initiator I/O Requests in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

SCSI Error Recovery in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding the sc_buf Structure in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

SCSI Adapter Device Driver.

The **close** subroutine, **ioctl or ioctlx** subroutine, **open, openx, or creat** subroutine, **read, readx, readv, or readvx** subroutine, **write, writex, writev, or writevx** subroutine.

The **cd** Special File, **omd** Special File, **rhdisk** Special File.

---

scsidisk SCSI Device Driver

Purpose

Supports the small computer system interface (SCSI), the Fibre Channel Protocol for SCSI (FCP), and the SCSI protocol over Internet (iSCSI) fixed disk, CD-ROM (compact disk read only memory), and read/write optical (optical memory) devices.

Syntax

```
#include <sys/devinfo.h>
#include <sys/scsi.h>
#include <sys/scdisk.h>
#include <sys/pcm.h>
#include <sys/mpio.h>
```

Device-Dependent Subroutines

Typical fixed disk, CD-ROM, and read/write optical drive operations are implemented using the **open**, **close, read**, **write**, and **ioctl** subroutines. The scsidisk device driver has additional support added for MPIO capable devices.

open and close Subroutines

The **open** subroutine applies a reservation policy based on the ODM **reserve_policy** attribute, previously the **open** subroutine always applied a SCSI2 reserve. The **open** and **close** subroutines will support working with multiple paths to a device if the device is a MPIO capable device.

The **openx** subroutine is intended primarily for use by the diagnostic commands and utilities. Appropriate authority is required for execution. If an attempt is made to run the **open** subroutine without the proper authority, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EPERM**.

The *ext* parameter passed to the **openx** subroutine selects the operation to be used for the target device. The **/usr/include/sys/scsi.h** file defines possible values for the *ext* parameter.

The *ext* parameter can contain any combination of the following flag values logically ORed together:

SC_DIAGNOSTIC Places the selected device in Diagnostic mode. This mode is singularly entrant; that is, only one process at a time can open it. When a device is in Diagnostic mode, SCSI operations are performed during **open** or **close** operations, and error logging is disabled. In Diagnostic mode, only the **close** and **ioctl** subroutine operations are accepted. All other device-supported subroutines return a value of -1 and set the **errno** global variable to a value of **EACCES**.

A device can be opened in Diagnostic mode only if the target device is not currently opened. If an attempt is made to open a device in Diagnostic mode and the target device is already open, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EACCES**.

SC_FORCED_OPEN Forces a target reset, regardless of whether another initiator has the device reserved. The target reset is sent to the device before the **open** sequence begins. In other respects, the **open** operation runs normally.
Note: Target reset will reset all luns on the SCSI ID.

SC_RETAIN_RESERVATION Retains the reservation of the device after a **close** operation by not issuing the release. This flag prevents other initiators from using the device unless they break the host machine's reservation.

SC_NO_RESERVE Prevents the reservation of a device during an **openx** subroutine call to that device. This operation is provided so a device can be controlled by two processors that synchronize their activity by their own software means.

SC_SINGLE Places the selected device in Exclusive Access mode. Only one process at a time can open a device in Exclusive Access mode.

A device can be opened in Exclusive Access mode only if the device is not currently open. If an attempt is made to open a device in Exclusive Access mode and the device is already open, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EBUSY**. If the **SC_DIAGNOSTIC** flag is specified along with the **SC_SINGLE** flag, the device is placed in Diagnostic mode.

FCP Options to the openx Subroutine in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts* gives more specific information on the **open** operations.

readx and writex Subroutines

The **readx** and **writex** subroutines provide additional parameters affecting the raw data transfer. These subroutines pass the *ext* parameter, which specifies request options. The options are constructed by logically ORing zero or more of the following values:

HWRELOC Indicates a request for hardware relocation (safe relocation only).
UNSAFEREL Indicates a request for unsafe hardware relocation.
WRITEV Indicates a request for write verification.

ioctl Subroutine

ioctl subroutine operations that are used for the **scsidisk** device driver are specific to the following categories:

- Fixed disk and read/write optical devices only
- CD-ROM devices only
- Fixed disk, CD-ROM, and read/write optical devices

Fixed Disk and Read/Write Optical Devices: The following **ioctl** operation is available for fixed disk and read/write optical devices only:

DKIOLWRSE Provides a means for issuing a **write** command to the device and obtaining the target-device sense data when an error occurs. If the **DKIOLWRSE** operation returns a value of -1 and the `status_validity` field is set to a value of **SC_SCSI_ERROR**, valid sense data is returned. Otherwise, target sense data is omitted.

The **DKIOLWRSE** operation is provided for diagnostic use. It allows the limited use of the target device while operating in an active system environment. The *arg* parameter to the **DKIOLWRSE** operation contains the address of an **scsi_rdwrt** structure. This structure is defined in the **/usr/include/sys/scsi_buf.h** file.

The **devinfo** structure defines the maximum transfer size for a **write** operation. If an attempt is made to transfer more than the maximum, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Fixed Disk, CD-ROM, and Read/Write Optical Devices: The following **ioctl** operations are available for fixed disk, CD-ROM, and read/write optical devices:

IOCINFO Returns the **devinfo** structure defined in the**/usr/include/sys/devinfo.h** file. The **IOCINFO** operation is the only operation defined for all device drivers that use the **ioctl** subroutine. The remaining operations discussed in this article are all specific to fixed disk, CD-ROM, and read/write optical devices.

DKIOLRDSE Provides a means for issuing a **read** command to the device and obtaining the target-device sense data when an error occurs. If the **DKIOLRDSE** operation returns a value of -1 and the `status_validity` field is set to a value of **SC_SCSI_ERROR**, valid sense data is returned. Otherwise, target sense data is omitted.

The **DKIOLRDSE** operation is provided for diagnostic use. It allows the limited use of the target device while operating in an active system environment. The *arg* parameter to the **DKIOLRDSE** operation contains the address of an **scsi_rdwrt** structure. This structure is defined in the **/usr/include/sys/scsi_buf.h** file.

The **devinfo** structure defines the maximum transfer size for a **read** operation. If an attempt is made to transfer more than the maximum, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

DKIOLCMD When the device has been successfully opened in the Diagnostic mode, the **DKIOLCMD** operation provides the means for issuing any SCSI command to the specified device. If the **DKIOLCMD** operation is issued when the device is not in Diagnostic mode, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EACCES**. The device driver performs no error recovery or logging on failures of this operation.

The SCSI status byte and the adapter status bytes are returned through the *arg* parameter, which contains the address of a **scsi_iocmd** structure (defined in the **/usr/include/sys/scsi_buf.h** file). If the **DKIOLCMD** operation fails, the subroutine returns a value of -1 and sets the **errno** global variable to a nonzero value. In this case, the caller should evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions should be taken.

The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

DKPMR Issues a SCSI prevent media removal command when the device has been successfully opened. This command prevents media from being ejected until the device is closed, powered off and then back on, or until a **DKAMR** operation is issued. The *arg* parameter for the **DKPMR** operation is null. If the **DKPMR** operation is successful, the subroutine returns a value of 0. If the device is a SCSI fixed disk, the **DKPMR** operation fails, and the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. If the **DKPMR** operation fails for any other reason, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EIO**.

DKAMR Issues an allow media removal command when the device has been successfully opened. As a result media can be ejected using either the drives eject button or the **DKEJECT** operation. The *arg* parameter for this ioctl is null. If the **DKAMR** operation is successful, the subroutine returns a value of 0. If the device is a SCSI fixed disk, the **DKAMR** operation fails, and the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. For any other failure of this operation, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EIO**.

DKEJECT Issues an eject media command to the drive when the device has been successfully opened. The *arg* parameter for this operation is null. If the **DKEJECT** operation is successful, the subroutine returns a value of 0. If the device is a SCSI fixed disk, the **DKEJECT** operation fails, and the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. For any other failure of this operation, the subroutine returns a value of -1 and sets the **errno** variable to a value of **EIO**.

DKFORMAT Issues a format unit command to the specified device when the device has been successfully opened.

If the *arg* parameter for this operation is null, the format unit sets the format options valid (FOV) bit to 0 (that is, it uses the drives default setting). If the *arg* parameter for the **DKFORMAT** operation is not null, the first byte of the defect list header is set to the value specified in the first byte addressed by the *arg* parameter. This allows the creation of applications to format a particular type of read/write optical media uniquely.

The driver initially tries to set the FmtData and CmpLst bits to 0. If that fails, the driver tries the remaining three permutations of these bits. If all four permutations fail, this operation fails, and the subroutine sets the **errno** variable to a value of **EIO**.

If the **DKFORMAT** operation is specified for a fixed disk, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. If the **DKFORMAT** operation is attempted when the device is not in Exclusive Access mode, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EACCES**. If the media is write-protected, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EWRPROTECT**. If the format unit exceeds its timeout value, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **ETIMEDOUT**. For any other failure of this operation, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EIO**.

DKAUDIO Issues play audio commands to the specified device and controls the volume on the device's output ports. Play audio commands include: play, pause, resume, stop, determine the number of tracks, and determine the status of a current audio operation. The **DKAUDIO** operation plays audio only through the CD-ROM drives output ports. The *arg* parameter of this operation is the address of a **cd_audio_cmds** structure, which is defined in the **/usr/include/sys/scdisk.h** file. Exclusive Access mode is required.

If **DKAUDIO** operation is attempted when the device's audio-supported attribute is set to No, the subroutine returns a value of -1 and sets the **errno**global variable to a value of **EINVAL**. If the **DKAUDIO** operation fails, the subroutine returns a value of -1 and sets the **errno** global variable to a nonzero value. In this case, the caller should evaluate the returned status bytes to determine why the operation failed and what recovery actions should be taken.

DK_CD_MODE Determines or changes the CD-ROM data mode for the specified device. The CD-ROM data mode specifies what block size and special file are used for data read across the SCSI bus from the device. The **DK_CD_MODE** operation supports the following CD-ROM data modes:

CD-ROM Data Mode 1
512-byte block size through both raw (**dev/rcd***) and block special (**/dev/cd***) files

CD-ROM Data Mode 2 Form 1
2048-byte block size through both raw (**dev/rcd***) and block special (**/dev/cd***) files

CD-ROM Data Mode 2 Form 2
2336-byte block size through the raw (**dev/rcd***) special file only

CD-DA (Compact Disc Digital Audio)
2352-byte block size through the raw (**dev/rcd***) special file only

DVD-ROM
2048-byte block size through both raw (**/dev/rcd***) and block special (**/dev/cd***) files

DVD-RAM
2048-byte block size through both raw (**/dev/rcd***) and block special (**/dev/cd***) files

DVD-RW
2048-byte block size through both raw (**/dev/rcd***) and block special (**/dev/cd***) files

The **DK_CD_MODE** *arg* parameter contains the address of the **mode_form_op** structure defined in the **/usr/include/sys/scdisk.h** file. To have the **DK_CD_MODE** operation determine or change the CD-ROM data mode, set the `action` field of the **change_mode_form** structure to one of the following values:

CD_GET_MODE
Returns the current CD-ROM data mode in the `cd_mode_form` field of the **mode_form_op** structure, when the device has been successfully opened.

CD_CHG_MODE
Changes the CD-ROM data mode to the mode specified in the `cd_mode_form` field of the **mode_form_op** structure, when the device has been successfully opened in the Exclusive Access mode.

If a CD-ROM has not been configured for different data modes (via mode-select density codes), and an attempt is made to change the CD-ROM data mode (by setting the `action` field of the **change_mode_form**structure set to **CD_CHG_MODE**), the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. Attempts to change the CD-ROM mode to any of the DVD modes will also result in a return value of -1 and the **errno** global variable set to **EINVAL**.

If the **DK_CD_MODE** operation for **CD_CHG_MODE** is attempted when the device is not in Exclusive Access mode, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EACCES**. For any other failure of this operation, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EIO**.

DK_PASSTHRU

When the device has been successfully opened, the **DK_PASSTHRU** operation provides the means for issuing any SCSI command to the specified device. The device driver will perform limited error recovery if this operation fails. The **DK_PASSTHRU** operation differs from the **DKIOCMD** operation in that it does not require an **openx** command with the *ext* argument of **SC_DIAGNOSTIC**. Because of this, a **DK_PASSTHRU** operation can be issued to devices that are in use by other operations.

The SCSI status byte and the adapter status bytes are returned through the *arg* parameter, which contains the address of a **sc_passthru** structure (defined in the **/usr/include/sys/scsi.h** file). If the **DK_PASSTHRU** operation fails, the subroutine returns a value of -1 and sets the errno global variable to a nonzero value. If this happens the caller should evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions should be taken.

If a **DK_PASSTHRU** operation fails because a field in the **sc_passthru** structure has an invalid value, the subroutine will return a value of -1 and set the errno global variable to **EINVAL**. The **einval_arg** field will be set to the field number (starting with 1 for the version field) of the field that had an invalid value. A value of 0 for the **einval_arg** field indicates no additional information on the failure is available.

DK_PASSTHRU operations are further subdivided into requests which quiesce other I/O prior to issuing the request and requests that do not quiesce I/O. These subdivisions are based on the **devflags** field of the **sc_passthru** structure. When the **devflags** field of the **sc_passthru** structure has a value of **SC_MIX_IO**, the **DK_PASSTHRU** operation will be mixed with other I/O requests. **SC_MIX_IO** requests that write data to devices are prohibited and will fail. When this happens -1 is returned, and the errno global variable is set to **EINVAL**. When the **devflags** field of the **sc_passthru** structure has a value of **SC_QUIESCE_IO**, all other I/O requests will be quiesced before the **DK_PASSTHRU** request is issued to the device. If an **SC_QUIESCE_IO** request has its **timeout_value** field set to 0, the **DK_PASSTHRU** request will be failed with a return code of -1, the errno global variable will be set to **EINVAL**, and the **einval_arg** field will be set to a value of **SC_PASSTHRU_INV_TO** (defined in the **/usr/include/sys/scsi.h** file). If an **SC_QUIESCE_IO** request has a nonzero timeout value that is too large for the device, the **DK_PASSTHRU** request will be failed with a return code of -1, the errno global variable will be set to **EINVAL**, the **einval_arg** field will be set to a value of **SC_PASSTHRU_INV_TO** (defined in the **/usr/include/sys/scsi.h** file), and the **timeout_value** will be set to the largest allowed value.

The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum transfer size, the subroutine returns a value of -1, sets the errno global variable to a value of **EINVAL**, and sets the **einval_arg** field to a value of **SC_PASSTHRU_INV_D_LEN** (defined in the **/usr/include/sys/scsi.h** file).

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

DK_RWBUFFER When the device has been successfully opened, the **DK_RWBUFFER** operation provides the means for issuing one or more SCSI Write Buffer commands to the specified device. The device driver will perform full error recovery upon failures of this operation. The **DK_RWBUFFER** operation differs from the **DKIOCMD** operation in that it does not require an exclusive open of the device (for example, **openx** with the *ext* argument of **SC_DIAGNOSTIC**). Thus, a **DK_RWBUFFER** operation can be issued to devices that are in use by others. It can be used in conjunction with the **DK_PASSTHRU** ioctl, which (like **DK_RWBUFFER**) does not require an exclusive open of the device.

The *arg* parameter contains the address of a **sc_rwbuffer** structure (defined in the **/usr/include/sys/scsi.h** file). Before the **DK_RWBUFFER** ioctl is invoked, the fields of this structure should be set according to the desired behavior. The **mode** field corresponds to the **mode** field of the SCSI Command Descriptor Block (CDB) as defined in the *SCSI Primary Commands (SPC) Specification*. Supported modes are listed in the header file **/usr/include/sys/scsi.h**.

The device driver will quiesce all other I/O from the initiator issuing the Write Buffer ioctl until the entire operation completes. Once the Write Buffer ioctl completes, all quiesced I/O will be resumed.

The SCSI status byte and the adapter status bytes are returned through the *arg* parameter, which contains the address of a **sc_rwbuffer** structure (defined in the **/usr/include/sys/scsi.h** file). If the **DK_RWBUFFER** operation fails, the subroutine returns a value of -1 and sets the **errno** global variable to a nonzero value. In this case, the caller should evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions should be taken.

If a **DK_RWBUFFER** operation fails because a field in the **sc_rwbuffer** structure has an invalid value, the subroutine will return a value of -1 and set the **errno** global variable to **EINVAL**.

The **DK_RWBUFFER** ioctl allows the user to issue multiple SCSI Write Buffer commands (CDBs) to the device through a single ioctl invocation. This is useful for applications such as microcode download where the user provides a pointer to the entire microcode image, but, due to size restrictions of the device buffer(s), desires that the images be sent in fragments until the entire download is complete.

If the **DK_RWBUFFER** ioctl is invoked with the **fragment_size** member of the **sc_rwbuffer** struct equal to **data_length**, a single Write Buffer command will be issued to the device with the **buffer_offset** and **buffer_ID** of the SCSI CDB set to the values provided in the **sc_rwbuffer** struct.

If **data_length** is greater than **fragment_size** and **fragment_size** is a nonzero value, multiple Write Buffer commands will be issued to the device. The number of Write Buffer commands (SCSI CDBs) issued will be calculated by dividing the **data_length** by the desired **fragment_size**. This value will be incremented by 1 if the **data_length** is not an even multiple of **fragment_size**, and the final data transfer will be the size of this residual amount. For each Write Buffer command issued, the **buffer_offset** will be set to the value provided in the **sc_rwbuffer** struct (microcode downloads to SCSD devices requires this to be set to 0). For the first command issued, the **buffer_ID** will be set to the value provided in the **sc_rwbuffer** struct. For each subsequent Write Buffer command issued, the **buffer_ID** will be incremented by 1 until all fragments have been sent. Writing to noncontiguous **buffer_ID**s through a single **DK_RWBUFFER** ioctl is not supported. If this functionality is desired, multiple **DK_RWBUFFER** ioctls must be issued with the **buffer_ID** set appropriately for each invocation.
Note: No I/O is quiesced between ioctl invocations.

DK_RWBUFFER continued

If **fragment_size** is set to zero, an **errno** of **EINVAL** will be returned. If the desire is to send the entire buffer with one SCSI Write buffer command, this field should be set equal to **data_length**. An error of **EINVAL** will also be returned if the **fragment_size** is greater than the **data_length**.

The Parameter List Length (**fragment_size**) plus the Buffer Offset can not exceed the capacity of the specified buffer of the device. It is the responsibility of the caller of the Write Buffer ioctl to ensure that the **fragment_size** setting satisfies this requirement. A **fragment_size** larger than the device can accommodate will result in a SCSI error at the device, and the Write Buffer ioctl will simply report this error but take no action to recover.

The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum transfer size, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request sense data for a particular device.

DKPATHIOLCMD This command is only available for MPIO capable devices. The **DKPATHIOLCMD** command takes as input a pointer argument which points to a single **scsidisk_pathiocmd** structure. The **DKPATHIOLCMD** command behaves exactly like the**DKIOLCMD** command, except that the input path is used instead of the normal path selection. The **DKPATHIOLCMD** path is used for the **DKIOLCMD** command regardless of any path specified by a **DKPATHFORCE** ioctl command. A path cannot be unconfigured while it is being forced.

DKPATHFORCE This command is only available for MPIO capable devices. The **DKPATHFORCE** command takes as input a ushort path id. The path id should correspond to one of the path ids in CuPath ODM. The path id specifies a path to be used for all subsequent I/O commands, overriding any previous **DKPATHFORCE** path. A zero argument specifies that path forcing is terminated and that normal MPIO path selection is to be resumed. The PCM KE keeps track of the forcing of I/O on a path. The Device Driver is unaware of this state except.I/O commands sent in with the **DKPATHIOLCMD** command will override the **DKPATHFORCE** option and send the I/O down the path specified in **scsidisk_pathiocmd** structure

DKPATHRWBUFFER This command is only available for MPIO capable devices. The **DKPATHRWBUFFER** command takes as input a pointer argument which points to a single **scsidisk_pathiocmd** structure. The **DKPATHRWBUFFER** command behaves exactly like the **DKRWBUFFER** command, except that the input path is used rather than normal path selection. The **DKPATHRWBUFFER** path is used for the **DKRWBUFFER** command regardless of any path specified by a **DKPATHFORCE** ioctl command.

DKPATHPASSTHRU This command is only available for MPIO capable devices. The **DKPATHPASSTHRU** command takes as input a pointer argument which points to a single **scsidisk_pathiocmd** structure. The **DKPATHPASSTHRU** command behaves exactly like the **DKPASSTHRU** command, except that the input path is used rather than normal path selection. The **DKPATHPASSTHRU** path is used for the **DKPASSTHRU** command regardless of any path specified by a **DKPATHFORCE** ioctl command.

DKPCMPASSTHRU This command is only available for MPIO capable devices. The **DKPCMPASSTHRU** command takes as input a structure which is PCM specific, it is not defined by AIX. The PCM specific structure is passed to the PCM directly. This structure can be used to move information to or from a PCM.

Device Requirements

SCSI fixed disk, CD-ROM, and read/write optical drives have the following hardware requirements:

- SCSI fixed disks and read/write optical drives must support a block size of 512 bytes per block.
- If mode sense is supported, the write-protection (WP) bit must also be supported for SCSI fixed disks and read/write optical drives.
- SCSI fixed disks and read/write optical drives must report the hardware retry count in bytes 16 and 17 of the request sense data for recovered errors. If the fixed disk or read/write optical drive does not support this, the system error log may indicate premature drive failure.
- SCSI CD-ROM and read/write optical drives must support the 10-byte SCSI read command.

- SCSI fixed disks and read/write optical drives must support the SCSI write and verify command and the 6-byte SCSI write command.
- To use the **format** command operation on read/write optical media, the drive must support setting the format options valid (FOV) bit to 0 for the defect list header of the SCSI format unit command. If the drive does not support this, the user can write an application for the drive so that it formats media using the **DKFORMAT** operation.
- If a SCSI CD-ROM drive uses **CD_ROM Data Mode 1**, it must support a block size of 512 bytes per block.
- If a SCSI CD-ROM drive uses **CD_ROM data Mode 2 Form 1**, it must support a block size of 2048 bytes per block.
- If a SCSI CD-ROM drive uses **CD_ROM data Mode 2 Form 2**, it must support a block size of 2336 bytes per block.
- If a SCSI CD-ROM drive uses **CD_DA** mode, it must support a block size of 2352 bytes per block.
- To control volume using the **DKAUDIO** (play audio) operation, the device must support SCSI-2 mode data page 0xE.
- To use the **DKAUDIO** (play audio) operation, the device must support the following SCSI-2 optional commands:
  - read sub-channel
  - pause resume
  - play audio MSF
  - play audio track index
  - read TOC

Error Conditions

Possible **errno** values for **ioctl**, **open**,**read**, and **write** subroutines when using the **scsidisk** device driver include:

EACCES Indicates one of the following circumstances:
- An attempt was made to open a device currently open in Diagnostic or Exclusive Access mode.
- An attempt was made to open a Diagnostic mode session on a device already open.
- The user attempted a subroutine other than an **ioctl** or **close** subroutine while in Diagnostic mode.
- A **DKIOLCMD** operation was attempted on a device not in Diagnostic mode.
- **A DK_CD_MODE ioctl** subroutine operation was attempted on a device not in Exclusive Access mode.
- A **DKFORMAT** operation was attempted on a device not in Exclusive Access mode.

EBUSY Indicates one of the following circumstances:
- An attempt was made to open a session in Exclusive Access mode on a device already opened.
- The target device is reserved by another initiator.

EFAULT Indicates an illegal user address.

EFORMAT Indicates the target device has unformatted media or media in an incompatible format.

EINPROGRESS Indicates a CD-ROM drive has a play-audio operation in progress.

EINVAL Indicates one of the following circumstances:>

- A **DKAUDIO** (play-audio) operation was attempted for a device that is not configured to use the SCSI-2 play-audio commands.
- The **read** or **write** subroutine supplied an *nbyte* parameter that is not an even multiple of the block size.
- A sense data buffer length of greater than 255 bytes is not valid for a **DKIOLWRSE**, or **DKIOLRDSE ioctl** subroutine operation.
- The data buffer length exceeded the maximum defined in the **devinfo** structure for a **DKIOLRDSE**, **DKIOLWRSE**, or **DKIOLCMD ioctl** subroutine operation.
- An unsupported **ioctl** subroutine operation was attempted.
- An attempt was made to configure a device that is still open.
- An illegal configuration command has been given.
- A **DKPMR** (Prevent Media Removal), **DKAMR** (Allow Media Removal), or **DKEJECT** (Eject Media) command was sent to a device that does not support removable media.
- A **DKEJECT** (Eject Media) command was sent to a device that currently has its media locked in the drive.
- The data buffer length exceeded the maximum defined for a **strategy** operation.

EIO Indicates one of the following circumstances:

- The target device cannot be located or is not responding.
- The target device has indicated an unrecoverable hardware error.

EMEDIA Indicates one of the following circumstances:

- The target device has indicated an unrecoverable media error.
- The media was changed.

EMFILE Indicates an **open** operation was attempted for an adapter that already has the maximum permissible number of opened devices.

ENODEV Indicates one of the following circumstances:

- An attempt was made to access an undefined device.
- An attempt was made to close an undefined device.

ENOTREADY Indicates no media is in the drive.

ENXIO Indicates one of the following circumstances:

- The **ioctl** subroutine supplied an invalid parameter.
- A **read** or **write** operation was attempted beyond the end of the fixed disk.

EPERM Indicates the attempted subroutine requires appropriate authority.

ESTALE Indicates a read-only optical disk was ejected (without first being closed by the user) and then either reinserted or replaced with a second optical disk.

ETIMEDOUT Indicates an I/O operation has exceeded the given timer value.

EWRPROTECT Indicates one of the following circumstances:

- An **open** operation requesting **read/write** mode was attempted on read-only media.
- A **write** operation was attempted to read-only media.

Reliability and Serviceability Information

SCSI fixed disk devices, CD-ROM drives, and read/write optical drives return the following errors:

ABORTED COMMAND Indicates the device ended the command.

ADAPTER ERRORS Indicates the adapter returned an error.

GOOD COMPLETION Indicates the command completed successfully.

HARDWARE ERROR Indicates an unrecoverable hardware failure occurred during command execution or during a self-test.

ILLEGAL REQUEST	Indicates an illegal command or command parameter.
MEDIUM ERROR	Indicates the command ended with an unrecoverable media error condition.
NOT READY	Indicates the logical unit is offline or media is missing.
RECOVERED ERROR	Indicates the command was successful after some recovery was applied.
UNIT ATTENTION	Indicates the device has been reset or the power has been turned on.

Error Record Values for Media Errors

The fields defined in the error record template for fixed disk, CD-ROM, and read/write optical media errors are:

`Comment`	Indicates fixed disk, CD-ROM, or read/write optical media error.
`Class`	Equals a value of H, which indicates a hardware error.
`Report`	Equals a value of True, which indicates this error should be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error log entry should be created when this error occurs.
`Alert`	Equals a value of False, which indicates this error is not alertable.
`Err_Type`	Equals a value of Perm, which indicates a permanent failure.
`Err_Desc`	Equals a value of 1312, which indicates a disk operation failure.
`Prob_Causes`	Equals a value of 5000, which indicates media.
`User_Causes`	Equals a value of 5100, which indicates the media is defective.
`User_Actions`	Equals the following values: • 1601, which indicates the removable media should be replaced and retried • 00E1 Perform problem determination procedures
`Inst_Causes`	None.
`Inst_Actions`	None.
`Fail_Causes`	Equals the following values: • 5000, which indicates a media failure • 6310, which indicates a disk drive failure
`Fail_Actions`	Equals the following values: • 1601, which indicates the removable media should be replaced and retried • 00E1 Perform problem determination procedures
`Detail_Data`	Equals a value of 156, 11, HEX. This value indicates hexadecimal format.

Note: The `Detail_Data` field in the **err_rec** structure contains the **scsi_error_log_df** structure. The **err_rec**structure is defined in the **/usr/include/sys/errids.h** file. The **scsi_error_log_df** structure is defined in the **/usr/include/sys/scsi_buf.h** file.

The **scsi_error_log_df** structure contains the following fields:

`req_sense_data`
Contains the request-sense information from the particular device that had the error, if it is valid.

`dd1` Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.

`dd2` Contains the number of bytes read since the segment count was last increased.

`dd3` Contains the number of opens since the device was configured.

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Error Record Values for Hardware Errors

The fields defined in the error record template for fixed disk, CD-ROM, and read/write optical hardware errors, as well as hard-aborted command errors are:

`Comment`	Indicates fixed disk, CD-ROM, or read/write optical hardware error.
`Class`	Equals a value of H, which indicates a hardware error.
`Report`	Equals a value of True, which indicates this error should be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error log entry should be created when this error occurs.
`Alert`	Equal to a value of FALSE, which indicates this error is not alertable.
`Err_Type`	Equals a value of Perm, which indicates a permanent failure.
`Err_Desc`	Equals a value of 1312, which indicates a disk operation failure.
`Prob_Causes`	Equals a value of 6310, which indicates disk drive.
`User_Causes`	None.
`User_Actions`	None.
`Inst_Causes`	None.
`Inst_Actions`	None.
`Fail_Causes`	Equals the following values: • 6310, which indicates a disk drive failure • 6330, which indicates a disk drive electronics failure
`Fail_Actions`	Equals a value of 00E1, which indicates problem-determination procedures should be performed.
`Detail_Data`	Equals a value of 156, 11, HEX. This value indicates hexadecimal format. **Note:** The `Detail_Data` field in the **err_rec** structure contains the **scsi_error_log_df** structure. The **err_rec**structure is defined in the **/usr/include/sys/errids.h** file. The **scsi_error_log_df** structure is defined in the **/usr/include/sys/scsi_buf.h** file. The **scsi_error_log_df** structure contains the following fields: **`req_sense_data`** Contains the request-sense information from the particular device that had the error, if it is valid. **`dd1`** Contains the segment count, which is the number of megabytes read from the device at the time the error occurred. **`dd2`** Contains the number of bytes read since the segment count was last increased. **`dd3`** Contains the number of opens since the device was configured.

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Error Record Values for Adapter-Detected Hardware Failures

The fields defined in the error record template for fixed disk, CD-ROM, and read/write optical media errors adapter-detected hardware errors are:

`Comment`	Indicates adapter-detected fixed disk, CD-ROM, or read/write optical hardware failure.
`Class`	Equals a value of H, which indicates a hardware error.
`Report`	Equals a value of True, which indicates this error should be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error-log entry should be created when this error occurs.
`Alert`	Equal to a value of FALSE, which indicates this error is not alertable.
`Err_Type`	Equals a value of Perm, which indicates a permanent failure.

Err_Desc	Equals a value of 1312, which indicates a disk operation failure.
Prob_Causes	Equals the following values: • 3452, which indicates a device cable failure • 6310, which indicates a disk drive failure
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equals the following values: • 3452, which indicates a storage device cable failure • 6310, which indicates a disk drive failure • 6330, which indicates a disk-drive electronics failure
Fail_Actions	Equals a value of 0000, which indicates problem-determination procedures should be performed.
Detail_Data	Equals a value of 156, 11, HEX. This value indicates hexadecimal format. **Note:** The Detail_Data field in the **err_rec** structure contains the **scsi_error_log_df** structure. The **err_rec**structure is defined in the **/usr/include/sys/errids.h** file. The **scsi_error_log_df** structure is defined in the **/usr/include/sys/scsi_buf.h** file. The **scsi_error_log_df** structure contains the following fields: **req_sense_data** Contains the request-sense information from the particular device that had the error, if it is valid. **dd1** Contains the segment count, which is the number of megabytes read from the device at the time the error occurred. **dd2** Contains the number of bytes read since the segment count was last increased. **dd3** Contains the number of opens since the device was configured.

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Error Record Values for Recovered Errors

The fields defined in the error record template for fixed disk, CD-ROM, and read/write optical media errors recovered errors are:

Comment	Indicates fixed disk, CD-ROM, or read/write optical recovered error.
Class	Equals a value of H, which indicates a hardware error.
Report	Equals a value of True, which indicates this error should be included when an error report is generated.
Log	Equals a value of True, which indicates an error log entry should be created when this error occurs.
Alert	Equal to a value of FALSE, which indicates this error is not alertable.
Err_Type	Equals a value of Temp, which indicates a temporary failure.
Err_Desc	Equals a value of 1312, which indicates a physical volume operation failure.
Prob_Causes	Equals the following values: • 5000, which indicates a media failure • 6310, which indicates a disk drive failure
User_Causes	Equals a value of 5100, which indicates media is defective.

User_Actions	Equals the following values: • 0000, which indicates problem-determination procedures should be performed • 1601, which indicates the removable media should be replaced and retried
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equals the following values: • 5000, which indicates a media failure • 6310, which indicates a disk drive failure
Fail_Actions	Equals the following values: • 1601, which indicates the removable media should be replaced and retried • 00E1 Perform problem determination procedures
Detail_Data	Equals a value of 156, 11, HEX. This value indicates hexadecimal format. **Note:** The Detail_Data field in the **err_rec** structure contains the **scsi_error_log_df** structure. The **err_rec**structure is defined in the **/usr/include/sys/errids.h** file. The **scsi_error_log_df** structure is defined in the **/usr/include/sys/scsi_buf.h** file. The **scsi_error_log_df** structure contains the following fields: **req_sense_data** Contains the request-sense information from the particular device that had the error, if it is valid. **dd1** Contains the segment count, which is the number of megabytes read from the device at the time the error occurred. **dd2** Contains the number of bytes read since the segment count was last increased. **dd3** Contains the number of opens since the device was configured.

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Error Record Values for Unknown Errors

The fields defined in the error record template for fixed disk, CD-ROM, and read/write optical media errors unknown errors are:

Comment	Indicates fixed disk, CD-ROM, or read/write optical unknown failure.
Class	Equals a value of H, which indicates a hardware error.
Report	Equals a value of True, which indicates this error should be included when an error report is generated.
Log	Equals a value of True, which indicates an error log entry should be created when this error occurs.
Alert	Equal to a value of FALSE, which indicates this error is not alertable.
Err_Type	Equals a value of Unkn, which indicates the type of error is unknown.
Err_Desc	Equals a value of FE00, which indicates an undetermined error.
Prob_Causes	Equals the following values: • 3300, which indicates an adapter failure • 5000, which indicates a media failure • 6310, which indicates a disk drive failure
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equals a value of FFFF, which indicates the failure causes are unknown.

`Fail_Actions` Equals the following values:

- 00E1 Perform problem determination procedures
- 1601, which indicates the removable media should be replaced and retried

`Detail_Data` Equals a value of 156, 11, HEX. This value indicates hexadecimal format. **Note:** The `Detail_Data` field in the **err_rec** structure contains the **scsi_error_log_df** structure. The **err_rec**structure is defined in the **/usr/include/sys/errids.h** file. The **scsi_error_log_df** structure is defined in the **/usr/include/sys/scsi_buf.h** file.

The **scsi_error_log_df** structure contains the following fields:

req_sense_data
Contains the request-sense information from the particular device that had the error, if it is valid.

dd1 Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.

dd2 Contains the number of bytes read since the segment count was last increased.

dd3 Contains the number of opens since the device was configured.

Refer to the *Small Computer System Interface (SCSI) Specification* for the format of the request-sense data for a particular device.

Special Files

The **scsidisk** SCSI device driver uses raw and block special files in performing its functions.

> **Attention:** Data corruption, loss of data, or loss of system integrity (system crash) will occur if devices supporting paging, logical volumes, or mounted file systems are accessed using block special files. Block special files are provided for logical volumes and disk devices and are solely for system use in managing file systems, paging devices, and logical volumes. These files should not be used for other purposes.

The special files used by the **scsidisk** device driver include the following (listed by type of device):

- Fixed disk devices:

/dev/rhdisk0, **/dev/rhdisk1**,..., **/dev/rhdisk***n* Provide an interface to allow SCSI device drivers character access (raw I/O access and control functions) to SCSI fixed disks.

/dev/hdisk0, **/dev/hdisk1**,..., **/dev/hdisk***n* Provide an interface to allow SCSI device drivers block I/O access to SCSI fixed disks.

- CD-ROM devices:

/dev/rcd0, **/dev/rcd1**,..., **/dev/rcd***n* Provide an interface to allow SCSI device drivers character access (raw I/O access and control functions) to SCSI CD-ROM disks.

/dev/cd0, **/dev/cd1**,..., **/dev/cd***n* Provide an interface to allow SCSI device drivers block I/O access to SCSI CD-ROM disks.

- Read/write optical devices:

/dev/romd0, **/dev/romd1**,..., **/dev/romd***n* Provide an interface to allow SCSI device drivers character access (raw I/O access and control functions) to SCSI read/write optical devices.

/dev/omd0, **/dev/omd1**,..., **/dev/omd***n* Provide an interface to allow SCSI device drivers block I/O access to SCSI read/write optical devices.

–

Note: The prefix **r** on a special file name indicates the drive is accessed as a raw device rather than a block device. Performing raw I/O with a fixed disk, CD-ROM, or read/write optical drive requires that all data transfers be in multiples of the device block size. Also, all **lseek** subroutines that are made to the raw device driver must result in a file pointer value that is a multiple of the device block size.

Related Information

Special Files Overview in *AIX 5L Version 5.2 Files Reference*.

FCP Subsystem Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

A Typical Initiator-Mode FCP Driver Transaction Sequence in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Required FCP Adapter Device Driver ioctl Commands in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding the Execution of Initiator I/O Requests in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

FCP Error Recovery in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding the scsi_buf Structure in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

FCP Device Driver.

The **close** subroutine, **ioctl or ioctlx** subroutine, **open, openx, or creat** subroutine, **read, readx, readv, or readvx** subroutine, **write, writex, writev, or writevx** subroutine.

The **cd** Special File, **omd** Special File, **rhdisk** Special File.

---

rmt SCSI Device Driver

Purpose

Supports the sequential access bulk storage medium device driver.

Syntax

```
#include  <sys/devinfo.h>
#include  <sys/scsi.h>
#include  <sys/tape.h>
```

Note: The **/dev/rmt0** through **/dev/rmt255** special files provide access to magnetic tapes. Magnetic tapes are used primarily for backup, file archives, and other offline storage.

Device-Dependent Subroutines

Most tape operations are implemented using the **open**, **read**, **write**, and **close** subroutines. However, the **openx** subroutine must be used if the device is to be opened in Diagnostic mode.

open and close Subroutines

The **openx** subroutine is intended for use by the diagnostic commands and utilities. Appropriate authority is required for execution. Attempting to execute this subroutine without the proper authority causes the subroutine to return a value of -1 and sets the **errno** global variable to **EPERM**.

The **openx** subroutine allows the device driver to enter Diagnostic mode and disables command-retry logic. This action allows for execution of ioctl operations that perform special functions associated with diagnostic processing. Other **openx** capabilities, such as forced opens and retained reservations, are also available.

The *ext* parameter passed to the **openx** subroutine selects the operation to be used for the target device. The *ext* parameter is defined in the **/usr/include/sys/scsi.h** file. This parameter can contain any combination of the following flag values logically ORed together:

Flag Value	Description
SC_DIAGNOSTIC	Places the selected device in Diagnostic mode. This mode is singularly entrant. When a device is in Diagnostic mode, SCSI operations are performed during **open** or **close** operations and error logging is disabled. In Diagnostic mode, only the **close** and ioctl operations are accepted. All other device-supported subroutines return a value of -1, with the **errno** global variable set to a value of **EACCES**. A device can be opened in Diagnostic mode only if the target device is not currently opened. If an attempt is made to open a device in Diagnostic mode and the target device is already open, a value of -1 is returned and the **errno** global variable is set to **EACCES**.
SC_FORCED_OPEN	Forces a bus device reset (BDR) regardless of whether another initiator has the device reserved. The SCSI bus device reset is sent to the device before the **open** sequence begins. Otherwise, the **open** operation executes normally.
SC_RETAIN_RESERVATION	Retains the reservation of the device after a **close** operation by not issuing the release. This flag prevents other initiators from using the device unless they break the host machine's reservation.

″SCSI Options to the openx Subroutine″ in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts* gives more specific information on the open operations.

ioctl Subroutine

The **STIOCMD** ioctl operation provides the means for sending SCSI commands directly to a tape device. This allows an application to issue specific SCSI commands that are not directly supported by the tape device driver.

To use the **STIOCMD** operation, the device must be opened in Diagnostic mode. If this command is attempted while the device is not in Diagnostic mode, a value of -1 is returned and the **errno** global variable is set to a value of **EACCES**. The **STIOCMD** operation passes the address of a **sc_iocmd** structure. This structure is defined in the **/usr/include/sys/scsi.h** file.

Refer to the *Small Computer System Interface (SCSI) Specification* for the applicable device for information on issuing the parameters.

Error Conditions

In addition to those errors listed, ioctl, **open**, **read**, and **write** subroutines against this device are unsuccessful in the following circumstances:

Error	Description
EACCES	Indicates that a diagnostic command was issued to a device not in Diagnostic mode.
EAGAIN	Indicates that an attempt was made to open a device that was already open.
EBUSY	Indicates that the target device is reserved by another initiator.
EINVAL	Indicates that a value of **O_APPEND** is supplied as the mode in which to open.
EINVAL	Indicates that the *nbyte* parameter supplied by a **read** or **write** operation is not a multiple of the block size.
EINVAL	Indicates that a parameter to an ioctl operation is not valid.
EINVAL	Indicates that the requested ioctl operation is not supported on the current device.
EIO	Indicates that the tape drive has been reset or that the tape has been changed. This error is returned on open if the previous operation to tape left the tape positioned beyond beginning of tape upon closing.
EIO	Indicates that the device could not space forward or reverse the number of records specified by the `st_count` field before encountering an EOM (end of media) or a file mark.
EMEDIA	Indicates that the tape device has encountered an unrecoverable media error.
EMFILE	Indicates that an **open** operation was attempted for a SCSI adapter that already has the maximum permissible number of open devices.
ENOTREADY	Indicates that there is no tape in the drive or the drive is not ready.
ENXIO	Indicates that there was an attempt to write to a tape that is at EOM.
EPERM	Indicates that this subroutine requires appropriate authority.
ETIMEDOUT	Indicates a command has timed out.
EWRPROTECT	Indicates an **open** operation requesting read/write mode was attempted on a read-only tape.
EWRPROTECT	Indicates that an ioctl operation that affects the media was attempted on a read-only tape.

Reliability and Serviceability Information

Errors returned from tape devices are as follows:

Error	Description
ABORTED COMMAND	Indicates the device ended the command.
BLANK CHECK	Indicates that a read command encountered a blank tape.
DATA PROTECT	Indicates that a write was attempted on a write-protected tape.
GOOD COMPLETION	Indicates that the command completed successfully.
HARDWARE ERROR	Indicates that an unrecoverable hardware failure occurred during command execution or during a self-test.
ILLEGAL REQUEST	Indicates an illegal command or command parameter.
MEDIUM ERROR	Indicates that the command terminated with a unrecovered media error condition. This condition may be caused by a tape flaw or a dirty head.
NOT READY	Indicates that the logical unit is offline.
RECOVERED ERROR	Indicates that the command was successful after some recovery was applied.
UNIT ATTENTION	Indicates the device has been reset or powered on.

Medium, hardware, and aborted command errors from the above list are to be logged every time they occur. The **ABORTED COMMAND** error may be recoverable, but the error is logged if recovery fails. For the **RECOVERED ERROR** and recovered **ABORTED COMMAND** error types, thresholds are maintained; when they are exceeded, an error is logged. The thresholds are then cleared.

Note: There are device-related adapter errors that are logged every time they occur.

Error Record Values for Tape Device Media Errors

The fields defined in the error record template for tape-device media errors are:

Field	Description
`Comment`	Equal to tape media error.
`Class`	Equal to H, indicating a hardware error.

Field	Description
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error is not alertable.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1332, indicating a tape operation failure.
Prob_Causes	Equal to 5003, indicating tape media.
User_Causes	Equal to 5100 and 7401, indicating a cause originating with the tape and defective media, respectively.
User_Actions	Equal to 1601 and 0000, indicating respectively that the removable media should be replaced and the operation retried, and that problem determination procedures should be performed.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 5003, indicating tape media.
Fail_Actions	Equal to 1601 and 0000, indicating respectively that the removable media should be replaced and the operation retried and that problem determination procedures should be performed.

The Detail_Data field contains the command type, device and adapter status, and the request-sense information from the particular device in error. The Detail_Data field is contained in the **err_rec** structure. This structure is defined in the **/usr/include/sys/errids.h** file. The **sc_error_log_df** structure, which describes information contained in the Detail_Data field, is defined in the **/usr/include/sys/scsi.h** file.

Refer to the *Small Computer System Interface (SCSI) Specification* for the applicable device for the format of the particular request-sense information.

Error-Record Values for Tape or Hardware Aborted Command Errors

The fields in the **err_hdr** structure, as defined in the **/usr/include/sys/erec.h** file for hardware errors and aborted command errors, are:

Field	Description
Comment	Equal to a tape hardware or aborted command error.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	FALSE, indicating this error is not alertable.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1331, indicating a tape drive failure.
Prob_Causes	Equal to 6314, indicating a tape drive error.
User_Causes	None.
User_Actions	Equal to 0000, indicating that problem determination procedures should be performed.
Inst_Actions	None.
Fail_Causes	Equal to 5003 and 6314, indicating the failure cause is the tape and the tape drive, respectively.
Fail_Actions	Equal to 0000 to perform problem determination procedures.

The Detail_Data field contains the command type, device and adapter status, and the request-sense information from the particular device in error. The Detail_Data field is contained in the **err_rec** structure. This structure is defined in the **/usr/include/sys/errids.h** file. The **sc_error_log_df** structure, which describes information contained in the Detail_Data field, is defined in the **/usr/include/sys/scsi.h** file.

Refer to the *Small Computer System Interface (SCSI) Specification* for the applicable device for the format of the particular request-sense information.

Error-Record Values for Tape-Recovered Error Threshold Exceeded

The fields defined in the **err_hdr** structure, as defined in the **/usr/include/sys/erec.h** file for recovered errors that have exceeded the threshold counter, are:

Field	Description
Comment	Indicates the tape-recovered error threshold has been exceeded.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error is not alertable.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1331, indicating a tape drive failure.
Prob_Causes	Equal to 5003 and 6314, indicating the probable cause is the tape and tape drive, respectively.
User_Causes	Equal to 5100 and 7401, indicating that the media is defective and the read/write head is dirty, respectively.
User_Actions	Equal to 1601 and 0000, indicating that removable media should be replaced and the operation retried and that problem-determination procedures should be performed, respectively.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 5003 and 6314, indicating the cause is the tape and tape drive, respectively.
Fail_Actions	Equal to 0000, to perform problem determination procedures.

The Detail_Data field contains the command type, device and adapter status, and the request-sense information from the particular device in error. This field is contained in the **err_rec** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The Detail_Data field also specifies the error type of the threshold exceeded. The **sc_error_log_df** structure, which describes information contained in the Detail_Data field, is defined in the **/usr/include/sys/scsi.h** file.

Refer to the *Small Computer System Interface (SCSI) Specification* for the applicable device for the format of the particular request-sense information.

Error Record Values for Tape SCSI Adapter-Detected Errors

The fields in the **err_hdr** structure, as defined in the **/usr/include/sys/erec.h** file for adapter-detected errors, are:

Field	Description
Comment	Equal to a tape SCSI adapter-detected error.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error is not alertable.
Err_Type	Equal to PERM, indicating a permanent failure.
Err_Desc	Equal to 1331, indicating a tape drive failure.
Prob_Causes	Equal to 3300 and 6314, indicating an adapter and tape drive failure, respectively.
User_Causes	None.
User_Actions	Equal to 0000, indicating that problem determination procedures should be performed.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 3300 and 6314, indicating an adapter and tape drive failure, respectively.
Fail_Actions	Equal to 0000, to perform problem-determination procedures.

The Detail_Data field contains the command type and adapter status. This field is contained in the **err_rec** structure, which is defined by the **/usr/include/sys/err_rec.h** file. Request-sense information is not

available with this type of error. The **sc_error_log_df** structure describes information contained in the Detail_Data field and is defined in the **/usr/include/sys/scsi.h** file.

Refer to the *Small Computer System Interface (SCSI) Specification* for the applicable device for the format of the particular request-sense information.

Error-Record Values for Tape Drive Cleaning Errors

Some tape drives return errors when they need cleaning. Errors that occur when the drive needs cleaning are grouped under this class.

Field	Description
Comment	Indicates that the tape drive needs cleaning.
Class	Equal to H, indicating a hardware error.
Report	Equal to TRUE, indicating that this error should be included when an error report is generated.
Log	Equal to TRUE, indicating that an error-log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error is not alertable.
Err_Type	Equal to TEMP, indicating a temporary failure.
Err_Desc	Equal to 1332, indicating a tape operation error.
Prob_Causes	Equal to 6314, indicating that the probable cause is the tape drive.
User_Causes	Equal to 7401, indicating a dirty read/write head.
User_Actions	Equal to 0000, indicating that problem determination procedures should be performed.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 6314, indicating that the cause is the tape drive.
Fail_Actions	Equal to 0000, indicating to perform problem-determination procedures.

The Detail_Data field contains the command type and adapter status and also the request-sense information from the particular device in error. This field is contained in the **err_rec** structure, which is defined by the **/usr/include/sys/errids.h** file. The **sc_error_log_df** structure describes information contained in the Detail_Data field and is defined in the **/usr/include/sys/scsi.h** file.

Refer to the *Small Computer System Interface (SCSI) Specification* for the applicable device for the format of the particular request-sense information.

Error-Record Values for Unknown Errors

Errors that occur for unknown reasons are grouped in this class. Data-protect errors fall into this class. These errors, detected by the tape device driver, are never seen at the tape drive.

The **err_hdr** structure for unknown errors describes the following fields:

Field	Description
Comment	Equal to tape unknown error.
Class	Equal to all error classes.
Report	Equal to TRUE, indicating this error should be included when an error report is generated.
Log	Equal to TRUE, indicating an error-log entry should be created when this error occurs.
Alert	Equal to FALSE, indicating this error is not alertable.
Err_Type	Equal to UNKN, indicating the error type is unknown.
Err_Desc	Equal to 0xFE00, indicating the error description is unknown.
Prob_Causes	None.
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equal to 0xFFFF, indicating the failure cause is unknown.
Fail_Actions	Equal to 0000, indicating that problem-determination procedures should be performed.

The Detail_Data field contains the command type and adapter status, and the request- sense information from the particular device in error. The Detail_Data field is contained in the **err_rec** structure. This field is contained in the **/usr/include/sys/errids.h** file. The **sc_error_log_df** structure describes information contained in the Detail_Data field and is defined in the **/usr/include/sys/scsi.h** file.

Refer to the *Small Computer System Interface (SCSI) Specification* for the applicable device for the format of the particular request-sense information.

Files

/dev/rmt0, /dev/rmt0.1, /dev/rmt0.2, ..., /dev/rmt0.7,

/dev/rmt1, /dev/rmt1.1, /dev/rmt1.2, ..., /dev/rmt1.7,...,

/dev/rmt255, /dev/rmt255.1, /dev/rmt255.2, ..., /dev/rmt255.7	Provide an interface to allow SCSI device drivers to access SCSI tape drives.

Related Information

The **rhdisk** special file, **rmt** special file.

The **close** subroutine, **ioctl** subroutine, **open** subroutine, **openx** subroutine, **read** subroutine, **write** subroutine.

A Typical Initiator-Mode SCSI Driver Transaction Sequence in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Required SCSI Adapter Device Driver ioctl Commands in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding the Execution of Initiator I/O Requests in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

SCSI Error Recovery in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding the sc_buf Structure in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

SCSI Adapter Device Driver.

---

scsesdd SCSI Device Driver

Purpose

Device driver supporting the **SCSI Enclosure Services** device.

Syntax

```
#include <sys/devinfo.h>
#include <sys/scsi.h>
#include <sys/scses.h>
```

Description

The special files **/dev/ses0**, **/dev/ses1**, **...**, provide I/O access and control functions to the SCSI enclosure devices.

Typical SCSI enclosure services operations are implemented using the **open, ioctl**, and **close** subroutines.

Open places the selected **device** in Exclusive Access mode. This mode is singularly entrant; that is, only one process at a time can open it.

A **device** can be opened only if the device is not currently opened. If an attempt is made to open a **device** and the device is already open, a value of -1 is returned and the **errno** global variable is set to a value of **EBUSY**.

ioctl Subroutine

The following ioctl operations are available for **SCSI Enclosure Services** devices:

Operation	Description
IOCINFO	Returns the **devinfo** structure defined in the **/usr/include/sys/devinfo.h** file.
SESIOCMD	When the device has been successfully opened, this operation provides the means for issuing any SCSI command to the specified enclosure. The device driver performs no error recovery or logging-on failures of this ioctl operation. The SCSI status byte and the adapter status bytes are returned via the *arg* parameter, which contains the address of a **sc_iocmd** structure (defined in the **/usr/include/sys/scsi.h** file). If the **SESIOCMD** operation returns a value of -1 and the errno global variable is set to a nonzero value, the requested operation has failed. In this case, the caller should evaluate the returned status bytes to determine why the operation failed and what recovery actions should be taken. The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum, a value of -1 is returned and the errno global variable set to a value of **EINVAL.** Refer to the *Small Computer System Interface* (**SCSI**) *Specification* for the applicable device to get request sense information.

Device Requirements

The following hardware requirements exist for SCSI enclosure services devices:

- The device must support the SCSI-3 Enclosure Services Specification Revision 4 or later.
- The device can be addressed from a SCSI id different from the SCSI ids of the the SCSI devices inside the enclosure.
- The device must be ″well behaved″, when receiving SCSI inquiries to page code 0xC7. This means that if the device fails the inquiry to page code C7 with a check condition, then the check condition will be cleared by the next SCSI command. An explicit request sense is not required.
- If the device reports its ANSI version to be 3 (SCSI-3) in the standard inquiry data, then it must correctly reject all invalid requests for luns 8-31 (that is,the device cannot ignore the upper bits in Lun id and thus cannot treat Lun 8 as being Lun 0, etc).

Error Conditions

Ioctl and **open** subroutines against this device fail in the following circumstances:

Error	Description
EBUSY	An attempt was made to open a device already opened.
EFAULT	An illegal user address was entered.
EINVAL	The data buffer length exceeded the maximum defined in the **devinfo** structure for a **SESIOCMD** ioctl operation.

Error	Description
EINVAL	An unsupported ioctl operation was attempted.
EINVAL	An attempt was made to configure a device that is still open.
EINVAL	An illegal configuration command has been given.
EIO	The target device cannot be located or is not responding.
EIO	The target device has indicated an unrecovered hardware error.
EMFILE	An open was attempted for an adapter that already has the maximum permissible number of opened devices.
ENODEV	An attempt was made to access a device that is not defined.
ENODEV	An attempt was made to close a device that has not been defined.
ENXIO	The ioctl subroutine supplied an invalid parameter.
EPERM	The attempted subroutine requires appropriate authority.
ETIMEDOUT	An I/O operation has exceeded the given timer value.

Reliability and Serviceability Information

The following errors are returned from SCSI enclosure services devices:

Error	Description
ABORTED COMMAN	The device cancelled the command.
ADAPTER ERRORS	The adapter returned an error.
GOOD COMPLETION	The command completed successfully.
HARDWARE ERROR	An unrecoverable hardware failure occurred during command execution or during a self test.
ILLEGAL REQUEST	An illegal command or command parameter.
MEDIUM ERROR	The command terminated with a unrecovered media error condition.
NOT READY	The logical unit is off-line or media is missing.
RECOVERED ERROR	The command was successful after some recovery applied.
UNIT ATTENTION	The device has been reset or the power has been turned on.

Files

/dev/ses0,/dev/ses1...,/dev/sesn	Provides an interface to allow SCSI device drivers access to SCSI enclosure services devices.

Related Information

SCSI Subsystem Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

A Typical Initiator-Mode SCSI Driver Transaction Sequence in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Required SCSI Adapter Device Driver ioctl Commands in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding the Execution of Initiator I/O Requests in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

SCSI Error Recovery in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding the sc_buf Structure in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

SCSI Adapter Device Driver

Purpose

Supports the SCSI adapter.

Syntax

```
<#include /usr/include/sys/scsi.h>
<#include /usr/include/sys/devinfo.h>
```

Description

The **/dev/scsi***n* and **/dev/vscsi***n* special files provide interfaces to allow SCSI device drivers to access SCSI devices. These files manage the adapter resources so that multiple SCSI device drivers can access devices on the same SCSI adapter simultaneously. The **/dev/vscsi***n* special file provides the interface for the SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A, while the **/dev/scsi***n* special file provides the interface for the other SCSI adapters. SCSI adapters are accessed through the special files **/dev/scsi0**, **/dev/scsi1**, ... and **/dev/vscsi0**, **/dev/vscsi1**, ...

The **/dev/scsi***n* and **/dev/vscsi***n* special files provide interfaces for access for both initiator and target mode device instances. The host adapter is an initiator for access to devices such as disks, tapes, and CD-ROMs. The adapter is a target when accessed from devices such as computer systems, or other devices that can act as SCSI initiators.

Device-Dependent Subroutines

The SCSI adapter device driver supports only the **open**, **close**, and **ioctl** subroutines. The **read** and **write** subroutines are not supported.

open and close Subroutines

The **openx** subroutine provides an adapter diagnostic capability. The **openx** subroutine provides an *ext* parameter. This parameter selects the adapter mode and accepts the **SC_DIAGNOSTIC** value. This value is defined in the **/usr/include/sys/scsi.h** file and places the adapter in Diagnostic mode.

Note: Some of the SCSI adapter device driver's open and close subroutines do not support the diagnostic mode *ext* parameter. (**SC_DIAGNOSTIC**). If such an open is attempted, the subroutine returns a value of -1 and the **errno** global value is set to **EINVAL**. The standalone diagnostic package provides all diagnostic capability.

In Diagnostic mode, only the **close** subroutine and ioctl operations are accepted. All other valid subroutines to the adapter return a value of -1 and set the **errno** global variable to a value of **EACCES**. In Diagnostic mode, the SCSI adapter device driver can accept the following requests:

- Run various adapter diagnostic tests.
- Download adapter microcode.

The **openx** subroutine requires appropriate authority to run. Attempting to run this subroutine without the proper authority causes the subroutine to return a value of -1, and set the **errno** global variable value to **EPERM**. Attempting to open a device already opened for normal operation, or when another **openx** subroutine is in progress, causes the subroutine to return a value of -1, and set the **errno** global variable to a value of **EACCES**.

Any kernel process can open the SCSI adapter device driver in Normal mode. For Normal mode the *ext* parameter is set to 0. However, a non-kernel process must have at least **dev_config** authority to open the SCSI adapter device driver in Normal mode. Attempting to execute a normal **open** subroutine without the proper authority causes the subroutine to return a value of -1, and set the **errno** global variable to a value of **EPERM**.

ioctl Subroutine

Along with the **IOCINFO** operation, the SCSI device driver defines specific operations for devices in non-diagnostic and diagnostic mode.

The **IOCINFO** operation is defined for all device drivers that use the **ioctl** subroutine, as follows:

- The operation returns a **devinfo** structure. This structure is defined in the **/usr/include/sys/devinfo.h** file. The device type in this structure is **DD_BUS**, and the subtype is **DS_SCSI**. The `flags` field is not used and is set to 0. Diagnostic mode is not required for this operation.
- The **devinfo** structure includes unique data such as the card SCSI ID and the maximum initiator mode data transfer size allowed (in bytes). A calling SCSI device driver uses this information to learn the maximum transfer size allowed for a device it controls on the SCSI adapter. In this way, the SCSI device driver can control devices across various SCSI adapters, with each device possibly having a different maximum initiator mode transfer size.

SCSI ioctl Operations for Adapters in Non-Diagnostic mode: The non-diagnostic operations are SCSI adapter device driver functions, rather than general device driver facilities. SCSI adapter device driver ioctl operations require that the adapter device driver is not in diagnostic mode. If these operations are attempted while the adapter is in diagnostic mode, a value of -1 is returned and the **errno** global variable is set to a value of **EACCES**.

The following SCSI operations are for adapters in non-diagnostic mode:

Operation	Description
SCIODNLD	Provides the means to download microcode to the adapter. The IBM SCSI-2 Fast/Wide Adapter/A device driver does not support this operation. Microcode download for the Fast/Wide adapter is supported in the standalone diagnostics package only.
SCIOEVENT	Registers the selected SCSI device instance to receive asynchronous event notification.
SCIOGTHW	Allows the caller to verify SCSI adapter device driver support for gathered writes.
SCIOHALT	Aborts the current command (if there is one), clears the queue of any pending commands, and places the device queue in a halted state for a particular device.
SCIOINQU	Provides the means to issue an inquire command to a SCSI device.
SCIOREAD	Sends a single block read command to the selected SCSI device.
SCIORESET	Allows the caller to force a SCSI device to release all current reservations, clear all current commands, and return to an initial state.
SCIOSTART	Opens a logical path to a SCSI target device. The host SCSI adapter acts as an initiator.
SCIOSTARTTGT	Opens a logical path to a SCSI initiator device. The host SCSI adapter acts as a target.
SCIOSTOP	Closes the logical path to a SCSI target device, where the SCSI adapter acts as an initiator.
SCIOSTOPTGT	Closes the logical path to a SCSI initiator device, where the host SCSI adapter was acting as a target.
SCIOSTUNIT	Provides the means to issue a SCSI Start Unit command to a selected SCSI device.
SCIOTUR	Sends a Test Unit Ready command to the selected SCSI device.

SCSI ioctl Operations for Adapters in Diagnostic Mode: The following operations for the **ioctl** subroutine are allowed only when the adapter has been successfully opened in Diagnostic mode. If these commands are attempted for an adapter not in Diagnostic mode, a value of -1 is returned and the **errno** global variable is set to a value of **EACCES**.

Operation	Description
SCIODIAG	Provides the means to issue adapter diagnostic commands.

Operation	Description
SCIODNLD	Provides the means to download microcode to the adapter.
SCIOTRAM	Provides the means to issue various adapter commands to test the card DMA interface and buffer RAM.

Note: Some of the SCSI adapter device drivers do not support the diagnostic mode ioctl operations.

To allow these operations to be run on multiple SCSI adapter card interfaces, a special return value is defined. A return value of -1 with an **errno** value of **ENXIO** indicates that the requested **ioctl** subroutine is not applicable to the current adapter card. This return value should not be considered an error for commands that require Diagnostic mode for execution.

Summary of SCSI Error Conditions

Possible **errno** values for the adapter device driver are:

Value	Description
EACCES	Indicates that an **openx** subroutine was attempted while the adapter had one or more devices in use.
EACCES	Indicates that a subroutine other than **ioctl** or **close** was attempted while the adapter was in Diagnostic mode.
EACCES	Indicates that a call to the **SCIODIAG** command was attempted while the adapter was not in Diagnostic mode.
EBUSY	Indicates that a delete operation was unsuccessful. The adapter is still open.
EFAULT	Indicates that the adapter is registering a diagnostic error in response to the **SCIODIAG** command. The **SCIODIAG** resume option must be issued to continue processing.
EFAULT	Indicates that a severe I/O error has occurred during an **SCIODNLD** command. Discontinue operations to this card.
EFAULT	Indicates that a copy between kernel and user space failed.
EINVAL	Indicates an invalid parameter or that the device has not been opened.
EIO	Indicates an invalid command. A **SCIOSTART** operation must be executed prior to this command, or an invalid SCSI ID and LUN combination must be passed in.
EIO	Indicates that the command has failed due to an error detected on the adapter or the SCSI bus.
EIO	Indicates that the device driver was unable to pin code.
EIO	Indicates that a kernel service failed, or that an unrecoverable I/O error occurred.
ENOCONNECT	Indicates that a SCSI bus fault occurred.
ENODEV	Indicates that the target device cannot be selected or is not responding.
ENOMEM	Indicates that the command could not be completed due to an insufficient amount of memory.
ENXIO	Indicates that the requested ioctl is not supported by this adapter.
EPERM	Indicates that the caller did not have the required authority.
ETIMEDOUT	Indicates that a SCSI command or adapter command has exceeded the time-out value.

Reliability and Serviceability Information

Errors detected by the adapter device driver may be one of the following:

- Permanent adapter or system hardware errors
- Temporary adapter or system hardware errors
- Permanent unknown adapter microcode errors
- Temporary unknown adapter microcode errors
- Permanent unknown adapter device driver errors
- Temporary unknown adapter device driver errors
- Permanent unknown system errors

- Temporary unknown system errors
- Temporary SCSI bus errors

Permanent errors are either errors that cannot be retried or errors not recovered before a prescribed number of retries has been exhausted. Temporary errors are either noncatastrophic errors that cannot be retried or retriable errors that are successfully recovered before a prescribed number of retries has been exhausted.

Error-Record Values for Permanent Hardware Errors

The error record template for permanent hardware errors detected by the SCSI adapter device driver is described below. Refer to the **rc** structure for the actual definition of the detail data. The **rc** structure is defined in the **/usr/include/sys/scsi.h** file:

SCSI_ERR1:

Field	Description
Comment	Permanent SCSI adapter hardware error.
Class	H, indicating a hardware error.
Report	TRUE, indicating this error should be included when an error report is generated.
Log	TRUE, indicating an error log entry should be created when this error occurs.
Alert	FALSE, indicating this error is not alertable.
Err_Type	PERM, indicating a permanent failure.
Err_Desc	0x1010, indicating an adapter error.
Prob_Causes	The following: **0x3330** Adapter hardware **0x3400** Cable **0x3461** Cable terminator **0x6000** Device
Fail_Causes	The following: **0x3300** Adapter **0x3400** Cable loose or defective **0x6000** Device
Fail_Actions	The following: **0x000** Perform problem determination procedures. **0x0301** Check the cable and its connections.
Detail_Data1	108, 11, and HEX

Error-Record Values for Temporary Hardware Errors

The error record template for temporary hardware errors detected by the SCSI adapter device driver follows:

SCSI_ERR2:

Field	Description
Comment	Temporary SCSI adapter hardware error.
Class	H, indicating a hardware error.
Report	TRUE, indicating this error should be included when an error report is generated.
Log	TRUE, indicating an error-log entry should be created when this error occurs.
Alert	FALSE, indicating this error is not alertable.
Err_Type	TEMP, indicating a temporary failure.
Err_Desc	0x1010, indicating an adapter error.

SCSI_ERR2:

Field	Description
Prob_Causes	The following: **0x3330** Adapter hardware **0x3400** Cable **0x3461** Cable terminator **0x6000** Device
Fail_Causes	The following: **0x3300** Adapter **0x3400** Cable loose or defective **0x6000** Device
Fail_Actions	The following: **0x000** Perform problem-determination procedures. **0x0301** Check the cable and its connections.
Detail_Data1	108, 11, and HEX

Error-Record Values for Permanent Unknown Adapter Microcode Errors

The error-record template for permanent unknown SCSI adapter microcode errors detected by the SCSI adapter device driver follows:

SCSI_ERR3:

Field	Description
Comment	Permanent SCSI adapter software error.
Class	H, indicating a hardware error.
Report	TRUE, indicating this error should be included when an error report is generated.
Log	TRUE, indicating an error log entry should be created when this error occurs.
Alert	FALSE, indicating this error is not alertable.
Err_Type	PERM, indicating a permanent failure.
Err_Desc	0x6100, indicating an adapter error.
Prob_Causes	0x3331, indicating an adapter microcode.
Fail_Causes	0x3300, indicating the adapter.
Fail_Actions	The following: **0x000** Perform problem determination procedures. **0x3301** If the problem persists (0x3000) contact the appropriate service representatives.
Detail_Data1	108, 11 and HEX

Error-Record Values for Temporary Unknown Adapter Microcode Errors

The error-record template for temporary unknown SCSI adapter microcode errors detected by the SCSI adapter device driver follows:

SCSI_ERR4:

Field	Description
Comment	Temporary unknown SCSI adapter software error.
Class	H.
Report	TRUE, indicating this error should be included when an error report is generated.
Log	TRUE, indicating an error log entry should be created when this error occurs.
Alert	FALSE, indicating this error is not alertable.
Err_Type	TEMP, indicating a temporary failure.
Err_Desc	Equal to 0x6100, indicating a microcode program error.
Prob_Causes	3331, indicating adapter microcode.
Fail_Causes	3300, indicating the adapter.

SCSI_ERR4:

Field	Description
Fail_Actions	The following: **0x000** Perform problem determination procedures. **0x3301** If the problem persists then (0x3000) contact the appropriate service representatives.
Detail_Data1	108, 11, and HEX

Error-Record Values for Permanent Unknown Adapter Device Driver Errors

The error-record template for permanent unknown SCSI adapter device driver errors detected by the SCSI adapter device driver follows:

SCSI_ERR5:

Field	Description
Comment	Permanent unknown driver error.
Class	S.
Report	TRUE, indicating this error should be included when an error report is generated.
Log	TRUE, indicating an error log entry should be created when this error occurs.
Alert	FALSE, indicating this error is not alertable.
Err_Type	PERM, indicating a permanent failure.
Err_Desc	0x2100, indicating a software program error.
Prob_Causes	0X1000, indicating a software program.
Fail_Causes	0X1000, indicating a software program.
Fail_Actions	0x3301, indicating that if the problem persists, then (0x3000) contact the appropriate service representatives.
Detail_Data1	108, 11, and HEX

Error-Record Values for Temporary Unknown Adapter Device Driver Errors

The error-record template for temporary unknown SCSI adapter device driver errors detected by the SCSI adapter device driver follows:

SCSI_ERR6:

Field	Description
Comment	Temporary unknown driver error.
Class	S.
Report	TRUE, indicating this error should be included when an error report is generated.
Log	TRUE, indicating an error log entry should be created when this error occurs.
Alert	FALSE, indicating this error is not alertable.
Err_Type	TEMP, indicating a temporary failure.
Err_Desc	0x2100, indicating a software program error.
Prob_Causes	0X1000, indicating a software program.
Fail_Causes	0X1000, indicating a software program.
Fail_Actions	0x3301, indicating that if the problem persists then (0x3000) contact the appropriate service representatives.
Detail_Data1	108, 11, and HEX

Error-Record Values for Permanent Unknown System Errors

The error-record template for permanent unknown system errors detected by the SCSI adapter device driver follows:

SCSI_ERR7:

Field	Description
Comment	Permanent unknown system error.
Class	H.

SCSI_ERR7:

Field	Description
Report	TRUE, indicating this error should be included when an error report is generated.
Log	TRUE, indicating an error log entry should be created when this error occurs.
Alert	FALSE, indicating this error is not alertable.
Err_Type	UNKN, indicating an unknown error.
Err_Desc	0xFE00, indicating an undetermined error.
Prob_Causes	0X1000, indicating a software program.
Fail_Causes	0X1000, indicating a software program.
Fail_Actions	0x0000 and 0x3301, indicating that problem-determination procedures should be performed; if the problem persists, then (0x3000) contact the appropriate service representatives.
Detail_Data1	108, 11, and HEX

Error-Record Values for Temporary Unknown System Errors

The error-record template for temporary unknown system errors detected by the SCSI adapter device driver follows:

SCSI_ERR8:

Field	Description
Comment	Temporary unknown system error.
Class	H.
Report	TRUE, indicating this error should be included when an error report is generated.
Log	TRUE, indicating an error log entry should be created when this error occurs.
Alert	FALSE, indicating this error is not alertable.
Err_Type	UNKN, indicating an unknown error.
Err_Desc	0xFE00, indicating an undetermined error.
Prob_Causes	0X1000, indicating a software program.
Fail_Causes	0X1000, indicating a software program.
Fail_Actions	0x0000 and 0x3301, indicating that problem-determination procedures should be performed; if the problem persists, then (0x3000) contact the appropriate service representatives.
Detail_Data1	108, 11, and HEX

Error-Record Values for Temporary SCSI Bus Errors

The error-record template for temporary SCSI bus errors by the SCSI adapter device driver follows:

SCSI_ERR10:

Field	Description
Comment	Temporary SCSI bus error.
Class	H, indicating a hardware error.
Report	True, indicating an error log entry should be created when this error occurs.
Alert	FALSE, indicating this error is not alertable.
Err_Type	TEMP, indicating a termporary failure.
Err_Desc	0x942, indicating a SCSI bus error.
Prob_Causes	The following: **0x3400** Cable **0x3461** Cable terminator **0x6000** Device **0x3300** Adapter Hardware
Fail_Causes	The following: **0x3400** Cable loose or defective **0x6000** Device **0x3300** Adapter

SCSI_ERR10:

Field	Description
Fail_Actions	The following: **0x000** Perform problem determination procedures. **0x0301** Check the cable and its connections.
Detail_Data	108, 11, and HEX.

Managing Dumps

The SCSI adapter device driver is a target for the system dump facility. The **DUMPINIT** and **DUMPSTART** options to the **dddump** entry point support multiple or redundant calls.

The **DUMPQUERY** option returns a minimum transfer size of 0 bytes and a maximum transfer size equal to the maximum transfer size supported by the SCSI adapter device driver.

To be processed, calls to the SCSI adapter device driver **DUMPWRITE** option should use the *arg* parameter as a pointer to the **sc_buf** structure. Using this interface, a SCSI **write** command can be run on a previously started (opened) target device. The *uiop* parameter is ignored by the SCSI adapter device driver. Spanned, or consolidated, commands are not supported using **DUMPWRITE**.

Note: The various **sc_buf** status fields, including the b_error field, are not set at completion of the **DUMPWRITE**. Error logging is, of necessity, not supported during the dump.

Successful completion of the **dddump** entry point is indicated by a 0. If unsuccessful, the entry point returns one of the following:

Value	Description
EINVAL	Indicates that the adapter device driver was passed a request that was not valid, such as attempting a **DUMPSTART** option before successfully executing a **DUMPINIT** option.
EIO	Indicates that the adapter device driver was unable to complete the command due to a lack of required resources or due to an I/O error.
ETIMEDOUT	Indicates that the adapter did not respond with status before the passed command time-out value expired.

Files

/dev/scsi0, **/dev/scsi1**,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.
/dev/vscsi0, **/dev/vscsi1,..., /dev/vscsi***n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **scdisk** SCSI device driver, **rmt** SCSI device driver, **tmscsi** SCSI device driver.

SCIOCMD SCSI Adapter Device Driver ioctl Operation

Purpose

Provides a means to issue any SCSI command to a SCSI device.

Description

The **SCIOCMD** operation allows the caller to issue a SCSI command to a selected adapter. This command can be used by system management routines to aid in the configuration of SCSI devices.

The *arg* parameter for the **SCIOCMD** operation is the address of a **sc_passthru** structure, which is defined in the **/usr/include/sys/scsi.h** field. The *sc_passthru* parameter allows the caller to select which SCSI and LUN IDS to send the command.

The SCSI status byte and the adapter status bytes are returned through the **sc_passthru** structure. If the **SCIOCMD** operation returns a value of -1 and the errno global variable is set to a nonzero value, the requested operation has failed. If this happens, the caller should evaluate the returned status bytes to determine why the operation failed and what recovery actions should be taken.

If the **SCIOCMD** operation fails because a field in the **sc_passthru** structure has an invalid value, the subroutine will return a value of -1, the errno global variable will be set to **EINVAL**, and the **einval_arg** field will be set to the field number (starting with 1 for the version field) of the field that had an invalid value. A value of 0 for the **einval_arg** field indicates no additional information is available.

The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum transfer size, the subroutine returns a value of -1, sets the errno global variable to a value of **EINVAL**, and sets the **einval_arg** field to a value of 18.

Refer to the *Small Computer System Interface (SCSI) Specification* to find out the format of the request-sense data for a particular device.

Return Values

The **SCIOCMD** operation returns a value of 0 when successfully completed. If unsuccessful, a value of -1 is returned, and the errno global variable is set to one of the following values:

EIO	A system error has occurred. Consider retrying the operation several (three) times, because another attempt may be successful. If an **EIO** error occurs and the **status_validity** field is set to **SC_SCSI_ERROR**, the **scsi_status** field has a valid value and should be inspected. If the **status_validity** field is zero and remains so on successive retries, an unrecoverable error has occurred. If the **status_validity** field is **SC_SCSI_ERROR** and the **scsi_status** field contains a *Check Condition* status, a SCSI request sense should be issued using the **SCIOCMD** ioctl to recover the sense data.
EFAULT	A user process copy has failed.
EINVAL	The device is not opened, or the caller has set a field in the **sc_passthru** structure to an invalid value.
EACCES	The adapter is in diagnostics mode.
ENOMEM	A memory request has failed.
ETIMEDOUT	The command has timed out. Consider retrying the operation several times, because another attempt may be successful.
ENODEV	The device is not responding.
ETIMEDOUT	The operation did not complete before the timeout value was exceeded.

Files

/dev/scsi0, /dev/scsi1, ... /dev/scsin	Provides an interface for all SCSI device drivers to access SCSI devices or adapters.

Related Information

"SCSI Adapter Device Driver" on page 294.

SCIODIAG (Diagnostic) SCSI Adapter Device Driver ioctl Operation

Purpose

Provides the means to issue adapter diagnostic commands.

Description

The **SCIODIAG** operation allows the caller to issue various adapter diagnostic commands to the selected SCSI adapter. These diagnostic command options are:

- Run the card Internal Diagnostics test
- Run the card SCSI Wrap test
- Run the card Read/Write Register test
- Run the card POS Register test
- Run the card SCSI Bus Reset test

An additional option allows the caller to resume the card Internal Diagnostics test from the point of a failure, which is indicated by the return value. The *arg* parameter for the **SCIODIAG** operation specifies the address of a **sc_card_diag** structure. This structure is defined in the **/usr/include/sys/scsi.h** file.

The actual adapter error-status information from each error reported by the card diagnostics is passed as returned parameters to the caller. Refer to the **sc_card_diag** structure defined in the **/usr/include/sys/scsi.h** file for the format of the returned data.

When the card diagnostics have completed (with previous errors), a value of **ENOMSG** is returned. At this point, no further **SCIODIAG** resume options are required, as the card internal diagnostics test has completed.

Adapter error status is always returned when a **SCIODIAG** operation results in an **errno** value of **EFAULT**. Because this error information is returned for each such volume, the final **ENOMSG** value returned for the card Internal Diagnostics test includes no error status information. Also, because this is a diagnostic command, these errors are not logged in the system error log.

Note: The SCSI adapter device driver performs no internal retries or other error-recovery procedures during execution of this operation. Error logging is also inhibited when running this command.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

Value	Description
EFAULT	Indicates that a bad copy between user and kernel space occurred.

Value	Description
EFAULT	For the integrated SCSI adapter on the 7008 and 7011 system models, this return value also indicates that the SCSI adapter device driver detected an error while attempting to run the **SCIODIAG** operation. In this case, the returned adapter status information must be analyzed to discover the cause of the error. Because this is a diagnostic command, this error is not logged in the system error log. For all other SCSI adapters, this value indicates that the card internal diagnostics have detected an error and paused. To continue, the caller must issue another **SCIODIAG** operation with the resume option. In response to this option, the card continues the diagnostics until either the end is reached or another error is detected. The caller must continue to issue **SCIODIAG** operations until the **EFAULT** error no longer returns.
EINVAL	Indicates a bad input parameter.
EIO	Indicates that the SCSI adapter device driver detected an error while attempting to run the **SCIODIAG** operation. In this case, the returned adapter status information must be analyzed to discover the cause of the error. Because this is a diagnostic command, this error is not logged in the system error log.
ENOMSG	Indicates that the card Internal Diagnostics test has completed.
ENXIO	Indicates that the operation or suboption selected is not supported on this adapter. This should not be treated as an error. The caller must check for this return value first (before checking for other **errno** values) to avoid mistaking this for a failing command.
ETIMEDOUT	Indicates that the adapter did not respond with status before the passed command time-out value expired. The **SCIODIAG** operation is a diagnostic command, so its errors are not logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices/adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

SCIODNLD (Download) SCSI Adapter Device Driver ioctl Operation

Purpose

Provides the means to download microcode to the adapter.

Description

The **SCIODNLD** operation provides for downloading microcode to the selected adapter. This operation can be used by system management routines to prepare the adapter for operation. The adapter can be opened in Normal or Diagnostic mode when the **SCIODNLD** operation is run.

There are two options for executing the **SCIODNLD** operation. The caller can either download microcode to the adapter or query the version of the currently downloaded microcode.

If the download microcode option is selected, a pointer to a download buffer and its length must be supplied in the caller's memory space. The maximum length of this microcode is adapter-dependent. If the adapter requires transfer of complete blocks, the microcode to be sent must be padded to the next largest block boundary. The block size, if any, is adapter-dependent. Refer to the reference manual for the particular SCSI adapter to find the adapter-specific requirements of the microcode buffer to be downloaded.

The SCSI adapter device driver validates the parameter values for such things as maximum length and block boundaries, as required. The *arg* parameter for the **SCIODNLD** operation specifies the address of a **sc_download** structure. This structure is defined in the **/usr/include/sys/scsi.h** file.

If the query version option is selected, the pointer and length fields in the passed parameter block are ignored. On successful completion of the **SCIODNLD** operation, the microcode version is contained in the `version_number` field.

The SCSI adapter device driver performs normal error-recovery procedures during execution of the **SCIODNLD** operation.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

Value	Description
EFAULT	Indicates that a severe I/O error has occurred, preventing completion of the download. In this case, further operations are not possible on the card, and the caller should discontinue commands to the card. The adapter error-status information is logged in the system error log.
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Indicates that the adapter device driver was unable to run the command due to incorrect input parameters. Check microcode length and block boundary for errors.
EIO	Indicates that the adapter device driver was unable to complete the command due to an unrecoverable I/O error or microcode cyclical redundancy check (CRC) error. If the card has on-board microcode, it may be able to continue running, and further commands may still be possible on this adapter. The adapter error-status information is logged in the system error log.
ENOMEM	Indicates insufficient memory is available to complete the command.
ENXIO	Indicates that the operation or suboption selected is not supported on this adapter and should not be treated as an error. The caller must check for this return value first (before checking for other **errno** values) to avoid mistaking this for a failing command.
ETIMEDOUT	Indicates that the adapter did not respond with status before the passed command time-out value expired. Since the download operation may not have completed, further operations on the card may not be possible. The caller should discontinue sending commands to the card. This error is also logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**,**...**, **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices and adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

SCSI Subsystem Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

SCIOEVENT (Event) SCSI Adapter Device Driver ioctl Operation

Purpose

Registers the selected SCSI device instance to receive asynchronous event notification.

Description

The **SCIOEVENT** operation registers the selected initiator or target-mode device for receiving asynchronous event notification. Only kernel mode processes or device drivers can call this function. If a user-mode process attempts an **SCIOEVENT** operation, the **ioctl** command is unsuccessful and the **errno** global value is set to **EPERM**.

The *arg* parameter to the **SCIOEVENT** operation should be set to the address of an **sc_event_struct** structure, which is in the **/usr/include/sys/scsi.h** file. If this is a target-mode instance, the **SCIOSTARTTGT** operation was used to open the device session; the caller then fills in the `ID` field with the SCSI ID of the SCSI initiator and sets the `logical unit number` (`LUN`) field to a value of 0. If this is an initiator-mode instance, the **SCIOSTART** operation was used to open the device session; the `ID` field is then set to the SCSI ID of the SCSI target, and the LUN is set to the LUN ID of the SCSI target. The device must have been previously opened using one of the start ioctls for this operation to succeed. If the device session is not opened, the **ioctl** command is unsuccessful and the returned **errno** global value is set to **EINVAL**.

The event registration performed by this ioctl is only allowed once per device session; only the first **SCIOEVENT** operation is accepted after the device is opened. Succeeding **SCIOEVENT** operations are unsuccessful, and the **errno** global value is set to **EINVAL**. The event registration is cancelled automatically when the device session is closed.

The caller fills in the `mode` field with one of the following values, which are defined in the **/usr/include/sys/scsi.h** file:

```
#define SC_IM_MODE   /* this is an initiator mode device */
#define SC_TM_MODE   /* this is a target mode device     */
```

The `async_func` field is filled in with the address of a pinned routine (in the calling program) that should be called by the SCSI adapter device driver whenever asynchronous event status is available for a registered device. The **struct sc_event_info** structure, defined in the **/usr/include/sys/scsi.h** file, is passed by address to the caller's **async_func** routine.

The `async_correlator` field can optionally be used by the caller to provide an efficient means of associating event information with the appropriate device. This field is saved by the SCSI adapter device driver and is returned, unchanged, with information passed back to the caller's **async_func** routine.

Reserved fields must be set to 0 by the caller.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

Value	Description
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Either an **SCIOSTART** or **SCIOSTARTTGT** operator has not been issued to this device instance, or this device is already registered for async events.
EPERM	Indicates the caller is not running in kernel mode, which is the only mode allowed to execute this operation.

Files

/dev/scsi0, /dev/scsi1,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.

/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

SCIOGTHW (Gathered Write) SCSI Adapter Device Driver ioctl Operation

Purpose

Allows the caller to verify that the SCSI adapter device driver to which this device instance is attached supports gathered writes.

Description

This operation allows the caller to verify that the gathered write function is supported by the SCSI adapter device driver before the caller attempts such an operation. The **SCIOGTHW** operation fails if a SCSI adapter device driver does not support gathered writes.

The *arg* parameter to the **SCIOGTHW** operation is set to null by the caller to indicate no input parameter is passed.

Note: This operation is not supported by all SCSI I/O Controllers. If not supported, **errno** is set to **EINVAL** and a value of -1 is returned.

Return Values

When completed successfully, the **SCIOGTHW** operation returns a value of 0, meaning gathered writes are supported. Otherwise, a value of -1 is returned and **errno** global variable is set to **EINVAL**.

Files

/dev/scsi0, **/dev/scsi1**,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.
/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

SCSI Adapter device driver.

SCIOHALT (Halt) SCSI Adapter Device Driver ioctl Operation

Purpose

Ends the current command (if there is one), clears the queue of any pending commands, and places the device queue in a halted state.

Description

The **SCIOHALT** operation allows the caller to end the current command (if there is one) to a selected device, clear the queue of any pending commands, and place the device queue in a halted state. The command causes the attached SCSI adapter to execute a SCSI abort message to the selected target device. This command is used by an upper-level SCSI device driver to end a running operation instead of waiting for the operation to complete or time out.

Once the **SCIOHALT** operation is sent, the calling device driver must set the **SC_RESUME** flag. This bit is located in the `flags` field of the next **sc_buf** structure to be processed by the SCSI adapter device driver. Any **sc_buf** structure sent without the **SC_RESUME** flag, after the device queue is in the halted state, is rejected.

The *arg* parameter to the **SCIOHALT** operation allows the caller to specify the SCSI identifier of the device to be reset. The least significant byte in the *arg* parameter is the LUN ID (logical unit number identifier) of the LUN on the SCSI controller to be halted. The next least significant byte is the SCSI ID. The remaining two bytes are reserved and must be set to a value of 0.

The SCSI adapter device driver performs normal error-recovery procedures during execution of this command. For example, if the abort message causes the SCSI bus to hang, a SCSI bus reset is initiated to clear the condition.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned, and the **errno** global variable is set to one of the following values:

Value	Description
EINVAL	Indicates a **SCIOSTART** operation was not issued prior to this operation.
EIO	Indicates an unrecoverable I/O error occurred. In this case, the adapter error-status information is logged in the system error log.
EIO	Indicates either the device is already stopping or the device driver was unable to pin code.
ENOCONNECT	Indicates a SCSI bus fault occurred.
ENODEV	Indicates the target SCSI ID could not be selected or is not responding. This condition is not necessarily an error and is not logged.
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates the adapter did not respond with status before the internal command time-out value expired. This error is logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**, **...**, **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices and adapters.
/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

SCIOINQU (Inquiry) SCSI Adapter Device Driver ioctl Operation

Purpose

Provides the means to issue an inquiry command to a SCSI device.

Description

The **SCIOINQU** operation allows the caller to issue a SCSI device inquiry command to a selected adapter. This command can be used by system management routines to aid in configuration of SCSI devices.

The *arg* parameter for the **SCIOINQU** operation is the address of an **sc_inquiry** structure. This structure is defined in the **/usr/include/sys/scsi.h** file. The **sc_inquiry** parameter block allows the caller to select the SCSI and LUN IDs to be queried.

The **SC_ASYNC** flag byte of the parameter block must not be set on the initial call to this operation. This flag is only set if a bus fault occurs and the caller intends to attempt more than one retry.

If successful, the returned inquiry data can be found at the address specified by the caller in the **sc_inquiry** structure. Successful completion occurs if a device responds at the requested SCSI ID, but the returned inquiry data must be examined to see if the requested LUN exists. Refer to the *Small Computer System Interface (SCSI) Specification* for the applicable device for the format of the returned data.

Note: The SCSI adapter device driver performs normal error-recovery procedures during execution of this command.

Return Values

When completed successfully this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

Value	Description
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Indicates that a **SCIOSTART** command was not issued prior to this command.
EIO	Indicates that an unrecoverable I/O error has occurred. If **EIO** is returned, the caller should retry the **SCIOINQU** operation since the first command may have cleared an error condition with the device. In case of an unrecovered error, the adapter error-status information is logged in the system error log.
ENOCONNECT	Indicates that a bus fault has occurred. The caller should respond by retrying with the **SC_ASYNC** flag set in the flag byte of the passed parameters. If more than one retry is attempted, only the last retry should be made with the **SC_ASYNC** flag set. Generally the SCSI adapter device driver cannot determine which device caused the SCSI bus fault, so this error is not logged.
ENODEV	Indicates that no SCSI controller responded to the requested SCSI ID. This return value implies that no LUNs exist on the requested SCSI ID. Therefore, when the **ENODEV** return value is encountered, the caller can skip this SCSI ID (and all LUNs on it) and go on to the next SCSI ID. This condition is not necessarily an error and is not logged.
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates that the adapter did not respond with a status before the internal command time-out value expired. On receiving the **ETIMEDOUT** return value, the caller should retry this command at least once, since the first command may have cleared an error condition with the device. This error is logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**, **...**, **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices/adapters.
/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

SCIOREAD (Read) SCSI Adapter Device Driver ioctl Operation

Purpose

Issues a single block SCSI **read** command to a selected SCSI device.

Description

The **SCIOREAD** operation allows the caller to issue a SCSI device **read** command to a selected adapter. System management routines use this command for configuring SCSI devices.

The *arg* parameter of the **SCIOREAD** operation is the address of an **sc_readblk** structure. This structure is defined in the **/usr/include/sys/scsi.h** header file.

This command results in the SCSI adapter device driver issuing a 6-byte format ANSI SCSI-1 **read** command. The command is set up to read only a single block. The caller supplies:

- Target device SCSI and LUN ID
- Logical block number to be read
- Length (in bytes) of the block on the device
- Time-out value (in seconds) for the command
- Pointer to the application buffer where the returned data is to be placed
- Flags parameter

The maximum block length for this command is 4096 bytes. The command will be rejected if the length is found to be larger than this value.

The **SC_ASYNC** flag of the flag parameter must not be set on the initial call to this operation. This flag is set only if a bus fault occurs and only if this is the caller's last retry attempt after this error occurs.

Note: The SCSI adapter device driver performs normal error-recovery procedures during execution of this command.

Return Values

When completed successfully this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

Value	Description
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Indicates that an **SCIOSTART** command was not issued prior to this command. If the **SCIOSTART** command was issued, then this indicates the block length field value is too large.
EIO	Indicates that an I/O error has occurred. If an **EIO** value is returned, the caller should retry the **SCIOREAD** operation since the first command may have cleared an error condition with the device. In the case of an adapter error, the system error log records the adapter error status information.
ENOCONNECT	Indicates that a bus fault has occurred. The caller should respond by retrying with the **SC_ASYNC** flag set in the flag byte of the passed parameters. If more than one retry is attempted, only the last retry should be made with the **SC_ASYNC** flag set. Generally, the SCSI adapter device driver cannot determine which device caused the bus fault, so this error is not logged.
ENODEV	Indicates that no SCSI controller responded to the requested SCSI ID. This return value implies that no logical unit numbers (LUNs) exist on the specified SCSI ID. This condition is not necessarily an error and is not logged.

Value	Description
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates the adapter did not respond with status before the internal time-out value expired. The caller should retry this command at least once, since the first command may have cleared an error condition with the device. The system error log records this error.

Files

/dev/scsi0, /dev/scsi1,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices/adapters.
/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

SCIORESET (Reset) SCSI Adapter Device Driver ioctl Operation

Purpose

Allows the caller to force a SCSI device to release all current reservations, clear all current commands, and return to an initial state.

Description

The **SCIORESET** operation allows the caller to force a SCSI device to release all current reservations, clear all current commands, and return to an initial state. This operation is used by system management routines to force a SCSI controller to release a competing SCSI initiator's reservation in a multi-initiator environment.

This operation actually executes a SCSI bus device reset (BDR) message to the selected SCSI controller on the selected adapter. The BDR message is directed to a SCSI ID. Therefore, all logical unit numbers (LUNs) associated with that SCSI ID are affected by the execution of the BDR.

For the operation to work effectively, a SCSI Reserve command should be issued after the **SCIORESET** operation through the appropriate SCSI device driver. Typically, the SCSI device driver open logic issues a SCSI Reserve command. This prevents another initiator from claiming the device.

There is a finite amount of time between the release of all reservations (by a **SCIORESET** operation) and the time the device is again reserved (by a SCSI Reserve command from the host). During this interval, another SCSI initiator can reserve the device instead. If this occurs, the SCSI Reserve command from this host fails and the device remains reserved by a competing initiator. The capability needed to prevent or recover from this event is beyond the SCSI adapter device driver and SCSI device driver components.

The *arg* parameter to the **SCIORESET** operation allows the caller to specify the SCSI ID of the device to be reset. The least significant byte in the *arg* parameter is the LUN ID of the LUN on the SCSI controller. The device indicated by the LUN ID should have been successfully started by a call to the **SCIOSTART** operation. The next least significant byte is the SCSI ID. The remaining two bytes are reserved and must be set to a value of 0.

Examples

1. The following example demonstrates actual use of this command. A SCSI ID of 1 is assumed, and an LUN of 0 exists on this SCSI controller.

```
open SCSI adapter device driver
SCIOSTART SCSI ID=1, LUN=0
SCIORESET SCSI ID=1, LUN=0 (to free any reservations)
SCIOSTOP SCSI ID=1, LUN=0
close SCSI adapter device driver
open SCSI device driver (normal open) for SCSI ID=1, LUN=0
...
Use device as normal
...
```

2. To make use of the **SC_FORCED_OPEN** flag of the SCSI device driver:

```
open SCSI device driver (with SC_FORCED_OPEN flag)
for SCSI ID=1, LUN=0
...
```

   Use the device as normal.

Both examples assume that the SCSI device driver **open** call executes a SCSI Reserve command on the selected device.

The SCSI adapter device driver performs normal error-recovery procedures during execution of this command. For example, if the BDR message causes the SCSI bus to hang, a SCSI bus reset will be initiated to clear the condition.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

Value	Description
EINVAL	Indicates an **SCIOSTART** command was not issued prior to this command.
EIO	Indicates an unrecoverable I/O error occurred. In this case, the adapter error-status information is logged in the system error log.
EIO	Indicates either the device is already stopping or the device driver is unable to pin code.
ENOCONNECT	Indicates that a bus fault has occurred. The caller should respond by retrying with the **SC_ASYNC** flag set in the flag byte of the passed parameters. If more than one retry is attempted, only the last retry should be made with the **SC_ASYNC** flag set. Generally, the SCSI adapter device driver cannot determine which device caused the bus fault, so this error is not logged in the system error log.
ENODEV	Indicates the target SCSI ID could not be selected or is not responding. This condition is not necessarily an error and is not logged.
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates the adapter did not respond with status before the internal command time-out value expired. This error is logged.

Files

/dev/scsi0, **/dev/scsi1**, **...**, **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.
/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

SCIOSTART (Start SCSI) Adapter Device Driver ioctl Operation

Purpose

Opens a logical path to a SCSI target device.

Description

The **SCIOSTART** operation opens a logical path to a SCSI device. The host SCSI adapter acts as an initiator device. This operation causes the adapter device driver to allocate and initialize the data areas needed to manage commands to a particular SCSI target.

The **SCIOSTART** operation must be issued prior to any of the other non-diagnostic mode operations, such as **SCIOINQU** and **SCIORESET**. However, the **SCIOSTART** operation is not required prior to calling the **IOCINFO** operation. Finally, when the caller is finished issuing commands to the SCSI target, the **SCIOSTOP** operation must be issued to release allocated data areas and close the path to the device.

The *arg* parameter to **SCIOSTART** allows the caller to specify the SCSI and LUN (logical unit number) identifier of the device to be started. The least significant byte in the *arg* parameter is the LUN, and the next least significant byte is the SCSI ID. The remaining two bytes are reserved and must be set to a value of 0.

Return Values

If completed successfully this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable set to one of the following values:

Value	Description
EIO	Indicates either an unrecoverable I/O error, or the device driver is unable to pin code.
EINVAL	Indicates either that the SCSI ID and LUN combination was incorrect (the combination may already be in use) or that the passed SCSI ID is the same as that of the adapter.

If the **SCIOSTART** operation is unsuccessful, the caller must not attempt other operations to this SCSI ID and LUN combination, since it is either already in use or was never successfully started.

Files

/dev/scsi0, **/dev/scsi1**, **...**, **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.
/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

SCIOSTARTTGT (Start Target) SCSI Adapter Device Driver ioctl Operation

Purpose

Opens a logical path to a SCSI initiator device.

Description

The **SCIOSTARTTGT** operation opens a logical path to a SCSI initiator device. The host SCSI adapter acts as a target. This operation causes the adapter device driver to allocate and initialize device-dependent information areas needed to manage data received from the initiator. It also makes the adapter device driver allocate system buffer areas to hold data received from the initiator. Finally, it makes the host adapter ready to receive data from the initiator.

This operation may only be called from a kernel process or device driver, as it requires that both the caller and the SCSI adapter device driver be able to directly access each other's code in memory.

Note: This operation is not supported by all SCSI I/O controllers. If not supported, **errno** is set to **ENXIO** and a value of -1 is returned.

The *arg* parameter to the **SCIOSTARTTGT** ioctl operation should be set to the address of an **sc_strt_tgt** structure, which is defined in the **/usr/include/sys/scsi.h** file. The caller fills in the ID field with the SCSI ID of the SCSI initiator and sets the `logical unit number (LUN)` field to `0`, as the initiator LUN is ignored for received data.

The caller sets the `buf_size` field to the desired size for all receive buffers allocated for this host target instance. This is an adapter-dependent parameter, which should be set to 4096 bytes for the SCSI I/O Controller. The `num_bufs` field is set to indicate how many buffers the caller wishes to have allocated for the device. This is also an adapter-dependent parameter. For the SCSI I/O Controller, it should be set to 16 or greater.

The caller fills in the `recv_func` field with the address of a pinned routine from its module, which the adapter device driver calls to pass received-data information structures. These structures tell the caller where the data is located and if any errors occurred.

The `tm_correlator` field can optionally be used by the caller to provide an efficient means of associating received data with the appropriate device. This field is saved by the SCSI adapter device driver and is returned, with information passed back to the caller's **recv_func** routine.

The `free_func` field is an output parameter for this operation. The SCSI adapter device driver fills this field with the address of a pinned routine in its module, which the caller calls to pass processed received-data information structures.

Currently, the host SCSI adapter acts only as LUN 0 when accessed from other SCSI initiators. This means the remotely-attached SCSI initiator can only direct data at one logical connection per host SCSI adapter. At most, only one calling process can open the logical path from the host SCSI adapter to a remote SCSI initiator. This does not prevent a single process from having multiple target devices opened simultaneously.

Note: Two or more SCSI target devices can have the same SCSI ID if they are physically attached to separate SCSI adapters.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

Value	Description
EINVAL	An **SCIOSTARTTGT** command has already been issued to this SCSI ID, the passed SCSI ID is the same as that of the adapter, the `LUN` field is not set to 0, the `buf_size` field is greater than 4096 bytes, the `num_bufs` field is less than 16, or the `recv_func` field is set to null.
EIO	Indicates an I/O error or kernel service failure occurred, preventing the device driver from enabling the selected SCSI ID.
ENOMEM	Indicates that a memory allocation error has occurred.
EPERM	Indicates the caller is not running in kernel mode, which is the only mode allowed to execute this operation.

Files

/dev/scsi0, /dev/scsi1,...,**/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.
/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

SCIOSTOP (Stop) Device SCSI Adapter Device Driver ioctl Operation

Purpose

Closes the logical path to a SCSI target device.

Description

The **SCIOSTOP** operation closes the logical path to a SCSI device. The host SCSI adapter acts as an initiator. The **SCIOSTOP** operation causes the adapter device driver to deallocate data areas allocated in response to a **SCIOSTART** operation. This command must be issued when the caller wishes to cease communications to a particular SCSI target. The **SCIOSTOP** operation should only be issued for a device successfully opened by a previous call to an **SCIOSTART** operation.

The **SCIOSTOP** operation passes the *arg* parameter. This parameter allows the caller to specify the SCSI and logical unit number (LUN) IDs of the device to be stopped. The least significant byte in the *arg* parameter is the LUN, and the next least significant byte is the SCSI ID. The remaining two bytes are reserved and must be set to 0.

Return Values

When completed successfully this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

Value	Description
EINVAL	Indicates that the device has not been opened. An **SCIOSTART** operation should be issued prior to calling the **SCIOSTOP** operation.
EIO	Indicates that the device drive was unable to pin code.

Files

/dev/scsi0, **/dev/scsi1**, **...**, **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.
/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

SCIOSTOPTGT (Stop Target) SCSI Adapter Device Driver ioctl Operation

Purpose

Closes a logical path to a SCSI initiator device.

Description

The **SCIOSTOPTGT** operation closes a logical path to a SCSI initiator device, where the host SCSI adapter acts as a target. This operation causes the adapter device driver to deallocate device-dependent information areas allocated in response to the **SCIOSTARTTGT** operation. It also causes the adapter device driver to deallocate system buffer areas used to hold data received from the initiator. Finally, it disables the host adapter's ability to receive data from the selected initiator.

This operation may only be called from a kernel process or device driver.

Note: This operation is not supported by all SCSI I/O Controllers. If not supported, **errno** is set to **ENXIO** and a value of -1 is returned.

The *arg* parameter to the **SCIOSTOPTGT** operation should be set to the address of an **sc_stop_tgt** structure, which is defined in the **/usr/include/sys/scsi.h** file. The caller fills in the `id` field with the SCSI ID of the initiator and sets the `logical unit number (LUN)` field to `0` as the initiator LUN is ignored for received data.

Note: The calling device driver should have previously freed any received-data areas by passing their information structures to the SCSI adapter device driver's **free_func** routine. All buffers allocated for this device are deallocated by the **SCIOSTOPTGT** operation regardless of whether the calling device driver has finished processing those buffers and has called the **free_func** routine.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

EINVAL	An **SCIOSTOPTGT** command has not been previously issued to this SCSI ID.
EPERM	Indicates the caller is not running in kernel mode, which is the only mode allowed to execute this operation.

Files

/dev/scsi0, /dev/scsi1, ...	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.

/dev/vscsi0, /dev/vscsi1, ...,**/dev/vscsi***n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

SCIOSTUNIT (Start Unit) SCSI Adapter Device Driver ioctl Operation

Purpose

Provides the means to issue a SCSI Start Unit command to a selected SCSI device.

Description

The **SCIOSTUNIT** operation allows the caller to issue a SCSI Start Unit command to a selected SCSI adapter. This command can be used by system management routines to aid in configuration of SCSI devices. For the **SCIOSTUNIT** operation, the *arg* parameter operation is the address of an **sc_startunit** structure. This structure is defined in the **/usr/include/sys/scsi.h** file.

The **sc_startunit** structure allows the caller to specify the SCSI and logical unit number (LUN) IDs of the device on the SCSI adapter that is to be started. The **SC_ASYNC** flag (in the flag byte of the passed parameter block) must not be set on the initial attempt of this command.

The `start_flag` field in the parameter block allows the caller to indicate the start option to the **SCIOSTUNIT** operation. When the `start_flag` field is set to TRUE, the logical unit is to be made ready for use. When FALSE, the logical unit is to be stopped.

> **Attention:** When the `immed_flag` field is set to TRUE, the SCSI adapter device driver allows simultaneous **SCIOSTUNIT** operations to any or all attached devices. It is important that when executing simultaneous SCSI Start Unit commands, the caller should allow a delay of at least 10 seconds between succeeding SCSI Start Unit command operations. The delay ensures that adequate power is available to devices sharing a common power supply. Failure to delay in this manner can cause damage to the system unit or to attached devices. Consult the technical specifications manual for the particular device and the appropriate hardware technical reference for your system.

The `immed_flag` field allows the caller to indicate the immediate option to the **SCIOSTUNIT** operation. When the `immed_flag` field is set to TRUE, status is to be returned as soon as the command is received by the device. When the field is set to FALSE, the status is to be returned after the operation is completed. The caller should set the `immed_flag` field to TRUE to allow overlapping **SCIOSTUNIT** operations to multiple devices on the SCSI bus. In this case, the **SCIOTUR** operation can be used to determine when the **SCIOSTUNIT** has actually completed.

Note: The SCSI adapter device driver performs normal error-recovery procedures during execution of the **SCIOSTUNIT** operation.

Return Values

When completed successfully, the **SCIOSTUNIT** operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

Value	Description
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Indicates that an **SCIOSTART** command was not issued prior to this command.

Value	Description
EIO	Indicates that an unrecoverable I/O error has occurred. If **EIO** is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. In case of an unrecovered error, the adapter error-status information is logged in the system error log.
ENOCONNECT	Indicates that a bus fault has occurred. The caller should respond by retrying with the **SC_ASYNC** flag set in the flag byte of the passed parameters. If more than one retry is attempted, only the last retry should be made with the **SC_ASYNC** flag set. Generally the SCSI adapter device driver cannot determine which device caused the SCSI bus fault, so this error is not logged.
ENODEV	Indicates that no SCSI controller responded to the requested SCSI ID. This condition is not necessarily an error and is not logged.
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates that the adapter did not respond with status before the internal command time-out value expired. If **ETIMEDOUT** is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. This error is logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.
/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*n*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

SCIOTRAM (Diagnostic) SCSI Adapter Device Driver ioctl Operation

Purpose

Provides the means to issue various adapter commands to test the card DMA interface and buffer RAM.

Description

The **SCIOTRAM** operation allows the caller to issue various adapter commands to test the card DMA interface and buffer RAM. The *arg* parameter block to the **SCIOTRAM** operation is the **sc_ram_test** structure. This structure is defined in the **/usr/include/sys/scsi.h** file and contains the following information:

- A pointer to a read or write test pattern buffer
- The length of the buffer
- An option field indicating whether a read or write operation is requested

Note: The SCSI adapter device driver is not responsible for comparing read data with previously written data. After successful completion of **write** or **read** operations, the caller is responsible for performing a comparison test to determine the final success or failure of this test.

The SCSI adapter device driver performs no internal retries or other error recovery procedures during execution of this operation. Error logging is inhibited when running this command.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to one of the following values:

Value	Description
EIO	Indicates that the adapter device driver detected an error. The specific adapter status is returned in the **sc_ram_test** parameter block. The SCIOTRAM operation is a diagnostic command and, as a result, this error is not logged in the system error log.
ENXIO	Indicates that the operation or suboption selected is not supported on this adapter. This should not be treated as an error. The caller must check for this return value first (before other **errno** values) to avoid mistaking this for a failing command.
ETIMEDOUT	Indicates the adapter did not respond with status before the passed command time-out value expired. The **SCIOTRAM** operation is a diagnostic command, so this error is not logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

SCIOTUR (Test Unit Ready) SCSI Adapter Device Driver ioctl Operation

Purpose

Sends a Test Unit Ready command to the selected SCSI device.

Description

The **SCIOTUR** operation allows the caller to issue a SCSI Test Unit Read (**SCIOSTUNIT**) command to a selected SCSI adapter. This command is used by system management routines to help configure SCSI devices.

The **sc_ready** structure allows the caller to specify the SCSI and the logical unit number (LUN) ID of the device on the SCSI adapter that is to receive the **SCIOTUR** operation. The **SC_ASYNC** flag (in the flag byte of the *arg* parameter block) must not be set during the initial attempt of this command. The **sc_ready** structure provides two output fields: `status_validity` and `scsi_status`. Using these two fields, the **SCIOTUR** operation returns the status to the caller. The *arg* parameter for the **SCIOTUR** operation specifies the address of the **sc_ready** structure, defined in the **/usr/include/sys/scsi.h** file.

When an **errno** value of **EIO** is received, the caller should evaluate the returned status in the `status_validity` and `scsi_status` fields. The `status_validity` field is set to the value **SC_SCSI_ERROR** to indicate that the `scsi_status` field has a valid SCSI bus status in it. The **/usr/include/sys/scsi.h** file contains typical values for the `scsi_status` field.

Following an **SCIOSTUNIT** operation, a calling program can tell by the SCSI bus status whether the device is ready. If an **errno** value of **EIO** is returned and the `status_validity` field is set to 0, an unrecovered error has occurred. If, on retry, the same result is obtained, the device should be skipped. If the `status_validity` field is set to **SC_SCSI_ERROR** and the `scsi_status` field indicates a Check Condition status, then another **SCIOTUR** command should be sent after a delay of several seconds.

After one or more attempts, the **SCIOTUR** operation should return a successful completion, indicating that the device was successfully started. If, after several seconds, the **SCIOTUR** operation still returns a `scsi_status` field set to a Check Condition status, the device should be skipped.

Note: The SCSI adapter device driver performs normal error-recovery procedures during execution of this command.

Return Values

When completed successfully, this operation returns a value of 0. For the **SCIOTUR** operation, this means the target device has been successfully started and is ready for data access. If unsuccessful, this operation returns a value of -1 and the **errno** global variable is set to one of the following values:

Value	Description
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Indicates the **SCIOSTART** operation was not issued prior to this command.
EIO	Indicates the adapter device driver was unable to complete the command due to an unrecoverable I/O error. If **EIO** is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. Following an unrecovered I/O error, the adapter error status information is logged in the system error log.
ENOCONNECT	Indicates a bus fault has occurred. The caller should retry after setting the **SC_ASYNC** flag in the flag byte of the passed parameters. If more than one retry is attempted, only the last retry should be made with the **SC_ASYNC** flag set. In general, the SCSI adapter device driver cannot determine which device caused the SCSI bus fault, so this error is not logged.
ENODEV	Indicates no SCSI controller responded to the requested SCSI ID. This condition is not necessarily an error and is not logged.
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates the adapter did not respond with a status before the internal command time-out value expired. If this return value is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. This error is logged in the system error log.

Files

/dev/scsi0, **/dev/scsi1**,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.
/dev/vscsi0, /dev/vscsi1,..., /dev/vscsi*m*	Provide an interface to allow SCSI-2 Fast/Wide Adapter/A and SCSI-2 Differential Fast/Wide Adapter/A device drivers to access SCSI devices or adapters.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

---

tmscsi SCSI Device Driver

Purpose

Supports processor-to-processor communications through the SCSI target-mode device driver.

Note: This operation is not supported by all SCSI I/O controllers.

Syntax

```
#include </usr/include/sys/devinfo.h>
#include </usr/include/sys/tmscsi.h>
#include </usr/include/sys/scsi.h>
```

Description

The Small Computer Systems Interface (SCSI) target-mode device driver provides an interface to allow processor-to-processor data transfer using the SCSI **send** command. This single device driver handles both SCSI initiator and SCSI target mode roles.

The user accesses the data transfer functions through the special files **/dev/tmscsi0.***xx*, **/dev/tmscsi1.***xx*, ... These are all character special files. The *xx* can be either **im**, initiator-mode interface, or **tm**, target-mode interface. The initiator-mode interface is used by the caller to transmit data, and the target-mode interface is used to receive data.

The least significant bit of the minor device number indicates to the device driver which mode interface is selected by the caller. When the least significant bit of the minor device number is set to a value of 1, the target-mode interface is selected. When the least significant bit is set to a value of 0, the initiator-mode interface is selected. For example, **tmscsi0.im** should be defined as an even-numbered minor device number to select the initiator-mode interface, and **tmscsi0.tm** should be defined as an odd-numbered minor device number to select the target-mode interface.

When the caller opens the initiator-mode special file a logical path is established, allowing data to be transmitted. The user-mode caller issues a **write**, **writev**, **writex**, or **writevx** system call to initiate data transmission. The kernel-mode user issues an **fp_write** or **fp_rwuio** service call to initiate data transmission. The SCSI target-mode device driver then builds a SCSI **send** command to describe the transfer, and the data is sent to the device. Once the write entry point returns, the calling program can access the transmit buffer.

When the caller opens the target-mode special file a logical path is established, allowing data to be received. The user-mode caller issues a **read**, **readv**, **readx**, or **readvx** system call to initiate data reception. The kernel-mode caller issues an **fp_read** or **fp_rwuio** service call to initiate data reception. The SCSI target-mode device driver then returns data received for the application.

The SCSI target mode device driver allows access as an initiator mode device through the **write** entry point. Target mode device access is made through the **read** entry point. Simultaneous access to the **read** and **write** entry points is possible by using two separate processes, one running **read** subroutines and the other running **write** subroutines.

The SCSI target mode device driver does not implement any protocol to manage the sending and receiving of data, with the exception of attempting to prevent an application from excessive received-data buffer usage. Any protocol required to maintain or otherwise manage the communications of data must be implemented in the calling program. The only delays in sending or receiving data through the target mode device driver are those inherent to the hardware and software driver environment.

Configuration Information

When the **tmscsi0** special file is configured, both the **tmscsi0.im** and **tmscsi0.tm** special files are created. An initiator-mode/target-mode pair for each device instance should exist, even if only one of the modes is being used. The target-mode SCSI ID for an attached device should be the same as the initiator-mode SCSI ID, but the logical unit number (LUN) is ignored in target mode, because the host SCSI adapter can only respond as LUN 0.

If multiple LUNs are supported on the attached initiator device, a pair of **tmscsi***n* special files (where *n* is the device instance) are generated for each SCSI ID/LUN combination. The initiator-mode special files allow simultaneous access to the associated SCSI ID/LUN combinations. However, only one of the target-mode special files for this SCSI ID can be opened at one time. This is because only one LUN 0 is supported on the host adapter and only one logical connection can be actively using this ID at one time. If a target-mode special file is open for a given SCSI ID, attempts to open other target-mode special files for the same ID will fail.

The target-mode device driver configuration entry point must be called only for the initiator-mode device number. The driver configuration routine automatically creates the configuration data for the target-mode device minor number based on the initiator-mode data.

Device-Dependent Subroutines

The target-mode device driver supports the **open**, **close**, **read**, **write**, **select**, and **ioctl** subroutines.

open Subroutine

The **open** subroutine allocates and initializes target or initiator device-dependent structures. No SCSI commands are sent to the device as a result of running the **open** subroutine.

The SCSI initiator or target-mode device must be configured and not already opened for that mode for the **open** subroutine to work. For the initiator-mode device to be successfully opened, its special file must be opened for writing only. For the target-mode device to be successfully opened, its special file must be opened for reading only.

Possible return values for the **errno** global variable include:

Value	Description
EAGAIN	Lock kernel service failed.
EBUSY	Attempted to execute an open for a device instance that is already open.
EINVAL	Attempted to execute an open for a device instance using an incorrect open flag, or device is not yet configured .
EIO	An I/O error occurred.
ENOMEM	The SCSI device is lacking memory resources.

close Subroutine

The **close** subroutine deallocates resources local to the target device driver for the target or initiator device. No SCSI commands are sent to the device as a result of running the **close** subroutine. Possible return values for the **errno** global variable include:

Value	Description
EINVAL	Attempted to execute a close for a device instance that is not configured.
EIO	An I/O error occurred.

read Subroutine

The **read** subroutine is supported only for the target-mode device. Data scattering is supported through the user-mode **readv** or **readvx** subroutine, or the kernel-mode **fp_rwuio** service call. If the **read** subroutine is unsuccessful, the return value is set to a return value of -1, and the **errno** global variable is set to the return value from the device driver. If the return value is something other than -1, then the read was successful and the return code indicates the number of bytes read. This should be validated by the caller. File offsets are not applicable and are therefore ignored for target-mode reads.

SCSI **send** commands provide the boundary for satisfying read requests. If more data is received in the **send** command than is requested in the current **read** operation, the requested data is passed to the caller, and the remaining data is retained and returned for the next **read** operation for this target device. If less data is received in the **send** command than is requested, the received data is passed for the read request, and the return value indicates how many bytes were read.

If a **send** command has not been completely received when a read request is made, the request blocks and waits for data. However, if the target device is opened with the **O_NDELAY** flag set, then the read

does not block; it returns immediately. If no data is available for the read request, the **read** is unsuccessful and the **errno** global variable is set to **EAGAIN**. If data is available, it is returned and the return value indicates the number of bytes received. This is true even if the **send** command for this data has not ended.

Note: Without the **O_NDELAY** flag set, the **read** subroutine can block indefinitely, waiting for data. Since the read data can come at any time, the device driver does not maintain an internal timer to interrupt the read. Therefore, if a time-out function is desired, it must be implemented by the calling program.

If the calling program wishes to break a blocked **read** subroutine, the program can generate a signal. The target-mode device driver receives the signal and ends the current **read** subroutine with failure. The **errno** global variable is then set to **EINTR**. The read returns with whatever data has been received, even if the **send** command has not completed. If and when the remaining data for the **send** command is received, it is queued, waiting for either another read request or a close. When the target receives the signal and the current read is returned, another read can be initiated or the target can be closed. If the read request that the calling program wishes to break completes before the signal is generated, the read completes normally and the signal is ignored.

The target-mode device driver attempts to queue received data ahead of requests from the application. A read-ahead buffer area (whose length is determined by the product of 4096 and the **num_bufs** attribute value in the configuration database) is used to store the queued data. As the application program executes **read** subroutines, the queued data is copied to the application data buffer and the read-ahead buffer space is again made available for received data. If an error occurs while copying the data to the caller's data buffer, the read fails and the **errno** global variable is set to **EFAULT**. If the **read** subroutines are not executed quickly enough, so that almost all the read-ahead buffers for the device are filled, data reception will be delayed until the application runs a **read** subroutine again. When enough area is freed, data reception is restored from the device. Data may be delayed, but it is not lost or ignored. If almost all the read-ahead buffers are filled, status information is saved indicating this condition. The application may optionally query this status through the **TMIOEVNT** operation. If the application uses the optional **select/poll** operation, it can receive asynchronous notification of this and other events affecting the target-mode instance.

The target-mode device driver handles only received data in its read entry point. All other initiator-sent SCSI commands are handled without intervention by the target-mode device driver. This also means the target-mode device driver does not directly generate any SCSI sense data or SCSI status.

The read entry point may optionally be used in conjunction with the select entry point to provide a means of asynchronous notification of received data on one or more target devices.

Possible return values for the **errno** global variable include:

Value	Description
EAGAIN	Indicates a non-blocking read request would have blocked, because no data is available.
EFAULT	An error occurred while copying data to the caller's buffer.
EINTR	Interrupted by a signal.
EINVAL	Attempted to execute a **read** for a device instance that is not configured, not open, or is not a target-mode minor device number.
EIO	I/O error occurred.

write Subroutine

The write entry point is supported only for the initiator-mode device driver. The write entry point generates a single SCSI **send** command in response to a calling program's write request. If the write request is for a length larger than the host SCSI adapter's maximum transfer length or if the request cannot be pinned as

a single request, then the **write** request fails with the **errno** global variable set to **EINVAL**. The maximum transfer size for this device is discovered by issuing an **IOCINFO ioctl** call to the target-mode device driver.

Some target mode capable adapters support data gathering of writes through the **user_mode writev** or **writevx** subroutine or the kernel-mode **fp_wruio** service call. The write buffers are gathered so that they are transferred, in order, as a single **send** command. The target-mode device driver passes information to the SCSI adapter device driver to allow it to perform the gathered write. Since the SCSI adapter device driver can be performing the gather function in software (when the hardware does not directly support data gathering), it is possible for the function to be unsuccessful because of a lack of memory or a copy error. The returned **errno** global variable is set to **ENOMEM** or **EFAULT**. Due to how gathered writes are handled, it is not possible for the target-mode device driver to perform retries. When an error does occur, the caller must retry or otherwise recover the operation.

If the **write** operation is unsuccessful, the return value is set to -1 and the **errno** global variable is set to the value of the return value from the device driver. If the return value is a value other than -1, the **write** operation was successful and the return value indicates the number of bytes written. The caller should validate the number of bytes sent to check for any errors. Since the entire data transfer length is sent in a single **send** command, a return code not equal to the expected total length should be considered an error. File offsets are not applicable and are ignored for target-mode writes.

If the calling program needs to break a blocked **write** operation, a signal should be generated. The target-mode device driver receives the signal and ends the current **write** operation. A **write** operation in progress fails, and the **errno** global variable is set to **EINTR**. The calling program may then continue by issuing another **write** operation, an **ioctl** operation, or may close the device. If the **write** operation the caller attempts to break completes before the signal is generated, the write completes normally and the signal is ignored.

The target-mode device driver automatically retries (up to the number of attempts specified by the value **TM_MAXRETRY** defined in the **/usr/include/sys/tmscsi.h** file) the **send** command if either a SCSI Busy response or no device response status is received for the command. By default, the target mode device driver delays each retry attempt by approximately two seconds to allow the target device to respond successfully. The caller can change the amount of time delayed through the **TMCHGIMPARM** operation. If retries are exhausted and the command is still unsuccessful, the write fails. The calling program can retry the **write** operation or perform other appropriate error recovery. All other error conditions are not retried but are returned with the appropriate **errno** global variable.

The target-mode device driver, by default, generates a time-out value, which is the amount of time allowed for the **send** command to complete. If the **send** command does not complete before the time-out value expires, the write fails. The time-out value is based on the length of the requested transfer, in bytes, and calculated as follows:

```
timeout_value = ((transfer_length / 65536) +1) *
10
```

In the calculation, 10 is the default scaling factor used to generate the time-out value. The caller can customize the time-out value through the **TMCHGIMPARM** operation.

One of the errors that can occur during a write is a SCSI status of check condition. A check-condition error requires a SCSI **request sense** command to be issued to the device. This returns the device's SCSI sense data, which must be examined to discover the exact cause of the check condition. To allow the target-mode device driver to work with a variety of target devices when in initiator mode, the device driver does not evaluate device sense data on check conditions. Therefore, the caller is responsible for evaluating the sense data to determine the appropriate error recovery. The **TMGETSENS** operation is provided to allow the caller to get the sense data. A unique **errno** global variable, **ENXIO**, is used to identify check conditions so that the caller knows when to issue the **TMGETSENS** operation. This error is not logged in the system error log by the SCSI device driver. The writer of the calling program must be

aware that according to SCSI standards, the **request sense** command must be the next command received by the device following a check-condition error. If any other command is sent to the device by this initiator, the sense data is cleared and the error information lost.

After each **write** subroutine, the target-mode device driver generates the appropriate return value and **errno** global variable. The device driver also updates a status area that is kept for the last command to each device. On certain errors, as well as successful completions, the caller may optionally read this status area to get more detailed error status for the command. The **TMIOSTAT** operation can be used for this purpose. The **errno** global variables covered by this status include **EIO**, **EBUSY**, **ENXIO**, and **ETIMEDOUT**.

Other possible return values for the **errno** global variable include:

Value	Description
EBUSY	SCSI reservation conflict detected. Try again later or make sure device reservation is ended before proceeding.
EFAULT	This is applicable only during data gathering. The **write** operation was unsuccessful due to a kernel service error.
EINTR	Interrupted by signal.
EINVAL	Attempted to execute a **write** operation for a device instance that is not configured, not open, or is not an initiator-mode minor device number. Transfer length too long, or could not pin entire transfer. Try command again with a smaller transfer length.
EIO	I/O error occurred. Either an unreproducible error occurred or retries were exhausted without success on an unreproducible error. Perform appropriate error recovery.
ENOCONNECT	Indicates a SCSI bus fault has occurred. The caller should respond by retrying with asynchronous data transfer allowed. This is accomplished by issuing a **TMIOASYNC** operation to this device prior to the retry. If more than one retry is attempted, the **TMIOASYNC** operation should be performed only before the last retry.
ENOMEM	This is applicable only during data gathering. The **write** operation was unsuccessful due to lack of system memory.
ENXIO	SCSI check condition occurred. Execute a **TMGETSENS** operation to get the device sense data and then perform required error recovery.
ETIMEDOUT	The command has timed out. Perform appropriate error recovery.

ioctl Subroutine

The following ioctl operations are provided by the target-mode device driver. Some are specific to either the target-mode device or the initiator-mode device. All require the respective device instance be open for the operation run.

Operation	Description
IOCINFO	Returns a structure defined in the **/usr/include/sys/devinfo.h** file.
TMCHGIMPARM	Allows the caller to change certain parameters used by the target mode device driver for a particular device instance.
TMGETSENS	Runs a SCSI **request sense** command and returns the sense data to the user.
TMIOASYNC	Allows succeeding initiator-mode commands to a particular target-mode device to use asynchronous data transfer.
TMIOCMD	Sends SCSI commands directly to the attached device.
TMIOEVNT	Allows the caller to query the device driver for status on certain events.
TMIORESET	Sends a Bus Device Reset message to an attached target-mode device.
TMIOSTAT	Allows the caller to get detailed status information about the previously-run **write** or **TMGETSENS** ioctl operation.

select Entry Point

The **select** entry point allows the caller to know when a specified event has occurred on one or more target-mode devices. The *events input* parameter allows the caller to specify which of one or more conditions it wants to be notified of by a bitwise OR of one or more flags. The target-mode device driver supports the following **select** events:

Event	Description
POLLIN	Check if received data is available.
POLLPRI	Check if status is available.
POLLSYNC	Return only events that are currently pending. No asynchronous notification occurs.

An additional event, **POLLOUT**, is not applicable and therefore is not supported by the target-mode device driver.

The *reventp output* parameter points to the result of the conditional checks. A bitwise OR of the following flags can be returned by the device driver:

Flag	Description
POLLIN	Received data is available.
POLLPRI	Status is available.

The *chan input* parameter is used for specifying a channel number. This is not applicable for non-multiplexed device drivers and should be set to a value of 0 for the target-mode device driver.

The **POLLIN** event is indicated by the device driver when any data is received for this target instance. A non-blocking **read** subroutine, if subsequently issued by the caller, returns data. For a blocking **read** subroutine, the read does not return until either the requested length is received or the **send** command completes, whichever comes first.

The **POLLPRI** event is indicated by the device driver when an exceptional event occurs. To determine the cause of the exceptional event, the caller must issue a **TMIOEVNT** operation to the device reporting the **POLLPRI** event.

The possible return value for the **errno** global variable includes:

Value	Description
EINVAL	A specified event is not supported, or the device instance is either not configured or not open.

Error Logging

Errors detected by the target-mode device driver can be one of the following:

- Unreproducible hardware error while receiving data
- Unreproducible hardware error during initiator command
- Unrecovered hardware error
- Recovered hardware error
- Device driver-detected software error

The target-mode device driver passes error-recovery responsibility for most detected errors to the caller. For these errors, the target-mode device driver does not know if this type of error is permanent or temporary. These types of errors are logged as temporary errors.

Only errors the target-mode device driver can itself recover through retries can be determined to be either temporary or permanent. The error is logged as temporary if it succeeds during retry (a recovered error) or as permanent if retries are unsuccessful (an unrecovered error). The return code to the caller indicates success if a recovered error occurs or failure if an unrecovered error occurs. The caller can elect to retry the command or operation, but the probability of retry success is low for unrecovered errors.

Related Information

The **tmscsi** special file.

The **errpt** command.

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver.

Error Logging Overview*Messages Guide and Reference*.

IOCINFO (Device Information) tmscsi Device Driver ioctl Operation

Purpose

Returns a structure defined in the **/usr/include/sys/devinfo.h** file.

Note: This operation is not supported by all SCSI I/O controllers.

Description

The **IOCINFO ioctl** operation returns a structure defined in the **/usr/include/sys/devinfo.h** header file. The caller supplies the address to an area of type `struct devinfo` in the *arg* parameter to the **IOCINFO** operation. The `device-type` field for this component is **DD_TMSCSI**; the `subtype` is **DS_TM**. The information returned includes the device's device dependent structure (DDS) information and the host SCSI adapter maximum transfer size for initiator-mode requests. The **IOCINFO** ioctl operation is allowed for both target and initiator modes. This command is not required for the caller, but it is useful for programs that need to know what the maximum transfer length is for **write** subroutines. It is also useful for calling programs that need the SCSI ID or logical unit number (LUN) of the device instance in use.

Files

/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

TMCHGIMPARM (Change Parameters) tmscsi Device Driver ioctl Operation

Purpose

Allows the caller to change parameters used by the target-mode device driver.

Note: This operation is not supported by all SCSI I/O controllers.

Description

The **TMCHGIMPARM** ioctl operation allows the caller to change certain parameters used by the target-mode device driver for a particular device instance. This operation is allowed only for the initiator-mode device. The *arg* parameter to the **TMCHGIMPARM** operation specifies the address of the **tm_chg_im_parm** structure defined in **/usr/include/sys/tmscsi.h** file.

Default values used by the device driver for these parameters usually do not require change. However, for certain calling programs, default values can be changed to fine-tune timing parameters related to error recovery.

The initiator-mode device must be open for this command to succeed. Once a parameter is changed through the **TMCHGIMPARM** operation, it remains changed until another **TMCHGIMPARM** operation is received or until the device is closed. At open time, these parameters are set to the default values.

Parameters that can be changed with this operation are the amount of delay (in seconds) between device driver-initiated retries of SCSI **send** commands and the amount of time allowed before the running of any **send** command times out. To indicate which of the possible parameters are being changed, the caller sets the appropriate bit in the `chg_option` field. Values of 0, 1, or multiple flags can be set in this field to indicate which parameters are being changed.

To change the delay between **send** command retries, the caller sets the **TM_CHG_RETRY_DELAY** flag in the `chg_option` field and places the desired delay value (in seconds) in the `new_delay` field of the structure. The retry delay can be changed with this command to any value between 0 and 255, inclusive, where 0 instructs the device driver to use as little delay as possible between retries. The default value is approximately 2 seconds.

To change the **send** command time-out value, the caller sets the **TM_CHG_SEND_TIMEOUT** flag in the `chg_option` field, sets the desired flag in the `timeout_type` field, and places the desired time-out value in the `new_timeout` field of the structure. A single flag must be set in the `time_out` field to indicate the desired form of the timeout. If the **TM_FIXED_TIMEOUT** flag is set in the `timeout_type` field, then the value placed in the `new_timeout` field is a fixed time-out value for all **send** commands. If the **TM_SCALED_TIMEOUT** flag is set in the `timeout_type` field, then the value placed in the `new_timeout` field is a scaling-factor used in the calculation for timeouts as shown under the description of the write entry point. The default **send** command time-out value is a scaled time-out with scaling factor of 10.

Regardless of the value of the `timeout_type` field, if the `new_timeout` field is set to a value of 0, the caller specifies ″no time out″ for the **send** command, allowing the command to take an indefinite amount of time. If the calling program wants to end a **write** operation, it generates a signal.

Files

/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, **tmscsi** SCSI device driver, SCSI Adapter device driver.

TMGETSENS (Request Sense) tmscsi Device Driver ioctl Operation

Purpose

Runs a SCSI **request sense** command and returns the sense data to the user.

Note: This operation is not supported by all SCSI I/O controllers.

Description

The **TMGETSENS** ioctl operation runs a SCSI **request sense** command and returns the sense data to the user. This operation is allowed only for the initiator-mode device. It is issued by the caller in response to a **write** subroutine **errno** global variable set to a value of **ENXIO**. This operation must be the next command issued to the device for this initiator or the sense data is lost. The *arg* parameter to the ioctl operation is the address of the **tm_get_sens** structure defined in the **/usr/include/sys/tmscsi.h** file. The caller must supply the address and length of a buffer used for holding the returned device-sense data in this structure. The maximum length for request-sense data is 255 bytes. The caller should refer to the SCSI specification for the target device to determine the correct length for the device's request-sense data. The lesser of either the sense data length requested or the actual sense data length is returned in the buffer passed by the caller. For the definition of the returned data, refer to the detailed SCSI specification for the device in use.

After each **TMGETSENS** operation, the target-mode device driver generates the appropriate **errno** global variable. If an error occurs, the return value is set to a value of -1 and the **errno** global variable is set to the value generated by the target-mode device driver. The device driver also updates a status area that is kept for the last command to each device. For certain errors, and upon successful completion, the caller can read this status area to get more detailed error status for the command. The **TMIOSTAT** operation can be used for this purpose. The **errno** global variables covered by this status include **EIO**, **EBUSY**, **ENXIO**, and **ETIMEDOUT**.

Files

/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

TMIOASYNC (Async) tmscsi Device Driver ioctl Operation

Purpose

Allows future initiator-mode commands for an attached target device to use asynchronous data transfer.

Note: This operation is not supported by all SCSI I/O controllers.

Description

The **TMIOASYNC** ioctl operation enables asynchronous data transfer for future initiator-mode commands on attached target devices. Only an initiator-mode device may use this operation. The *arg* parameter of the **TMIOASYNC** operation is set to a null value by the caller.

This operation is required when the caller is intending to retry a previous initiator SCSI command (other than those sent through the **TMIOCMD** operation) that was unsuccessful with a **SC_SCSI_BUS_FAULT**

status in the general_card_status field in the status structure returned by the **TMIOSTAT** operation. If more than one retry is attempted, this operation should be issued only before the last retry attempt.

This operation allows the device to run in asynchronous mode if the device does not negotiate for synchronous transfers. This operation affects all future initiator commands for this device. However, a SCSI reset or power-on to the device results in an attempt to again run synchronous data transfers. At open time, synchronous data transfers are attempted.

Files

/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

TMIOCMD (Direct) tmscsi Device Driver ioctl Operation

Purpose

Sends SCSI commands directly to the attached device.

Note: This operation is not supported by all SCSI I/O controllers.

Description

> **Attention:** The **TMIOCMD** operation is a very powerful operation. Extreme care must be taken by the caller before issuing any general SCSI command, as this may adversely affect the attached device, other SCSI devices on the SCSI bus, or even general system availability. It should only be used when no other means are available to run the required function or functions on the attached device. This operation requires at least **dev_config** authority to run.

The **TMIOCMD** operation provides a means of sending SCSI commands directly to the attached device. This operation is only allowed for the initiator-mode device. It enables a caller to issue specific SCSI commands that are not directly supported by the device driver. The caller is responsible for any and all error recovery associated with the sending of the SCSI command. No error recovery is performed by the device driver when the command is issued. The device driver does not log errors that occur while running the command.

The *arg* parameter to this command specifies the address of the **sc_iocmd** structure defined in the **/usr/include/sys/scsi.h** file. The caller fills in the SCSI command descriptor block area, command length (SCSI command block length), the time-out value for the command, and a flags field. If a data transfer is involved, the data length and buffer pointer areas, as well as the **B_READ** flag in the flags field, must be filled in. The **B_READ** is set to a value of 1 to indicate the command's data transfer is incoming, and **B_READ** is set to a value of 0 to indicate the data is outgoing. If there is no data transfer, these fields and flags are set to 0 values.

The target-mode device driver builds the appropriate command block to execute this operation, including ORing in the 3-bit logical unit number (LUN) identifier in the SCSI command based on the configuration information for this device instance. The returned **errno** global variable is generated and the status validity, SCSI bus status, and adapter status fields are updated to reflect the completion status for the command. These status areas are defined in the **/usr/include/sys/scsi.h** file.

Files

/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

TMIOEVNT (Event) tmscsi Device Driver ioctl Operation

Purpose

Allows the caller to query the device driver for event status.

Note: This operation is not supported by all SCSI I/O controllers.

Description

The **TMIOEVNT** ioctl operation allows the caller to query the device driver for status on certain events. The *arg* parameter to the **TMIOEVNT** operation specifies the address of the **tm_event_info** structure defined in the **/usr/include/sys/tmscsi.h** file. This operation conveys status that is generally not tied to a specific application program subroutine and would not otherwise be known to the application. For example, failure of an adapter function not associated directly with a SCSI command is reported through this facility.

Although this operation can be used independently of other commands to the target-mode device driver, it is most effective when issued in conjunction with the select entry point **POLLPRI** option. For this device driver, the **POLLPRI** option indicates an event has occurred that is reported through the **TMIOEVNT** operation. This allows the caller to be asynchronously notified of events occurring to the device instance, which means the **TMIOEVNT** operation need only be issued when an event occurs. Without the select entry point, it would be necessary for the caller to issue the **TMIOEVNT** operation after every **read** or **write** subroutine to know when an event has occurred. The select entry point allows the caller to monitor events on one or more target or initiator devices.

Because the caller is not generally aware of which adapter a particular device is attached to, event information in the **TMIOEVNT** operation is maintained for each device instance. Application programs should not view any information from one device's **TMIOEVNT** operation as necessarily affecting other devices opened through this device driver. Rather, the application must base its error recovery for each device on that device's particular **TMIOEVNT** information.

Event information is reported through the `events` field of the **tm_event_info** structure and can have the following values:

Value	Description
TM_FATAL_HDW_ERR	Adapter fatal hardware failure
TM_ADAP_CMD_FAILED	Unrecoverable adapter command failure
TM_SCSI_BUS_RESET	SCSI Bus Reset detected
TM_BUFS_EXHAUSTED	Maximum buffer usage detected

Some of the events that can be reported apply to any SCSI device, whether they are initiator-mode or target-mode devices. These events include **adapter fatal hardware failure**, **unrecoverable adapter command failure**, and **SCSI BUS Reset** detected. The **maximum buffer usage detected** event applies only to the target mode device and is never reported for an initiator-mode device instance.

The **adapter fatal hardware failure** event is intended to indicate a fatal condition. This means no further commands are likely to complete successfully to or from this SCSI device, as the adapter it is attached to has failed. In this case, the application should end the session with the device.

The **unrecoverable adapter command failure** event is not necessarily a fatal condition but can indicate that the adapter is not functioning properly. The application program has these possible actions:

- End the session with the device in the near future.
- End the session after multiple (two or more) such events.
- Attempt to continue the session indefinitely.

The **SCSI Bus Reset detection** event is mainly intended as information only but can be used by the application to perform further actions, if necessary. The Reset information can also be conveyed to the application during command execution, but the Reset must occur during the SCSI command for this to occur.

The **maximum buffer usage detected** event only applies to a given target-mode device; it is not be reported for an initiator device. This event indicates to the application that this particular target-mode device instance has filled its maximum allotted buffer space. The application should perform **read** subroutines fast enough to prevent this condition. If this event occurs, data is not lost, but it is delayed to prevent further buffer usage. Data reception is restored when the application empties enough buffers to continue reasonable operations. The **num_bufs** attribute may need to be increased from the default value to help minimize this problem.

Return Values

EFAULT	Operation failed due to a kernel service error.
EINVAL	Attempted to execute an ioctl operation for a device instance that is not configured, not open, or is not in the proper mode (initiator versus target) for this operation.
EIO	An I/O error occurred during the operation.
EPERM	For the **TMIOCMD** operation, the caller did not have **dev_config** authority.
ETIMEDOUT	The operation did not complete before the timeout expired.

Files

/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

TMIORESET (Reset Device) tmscsi Device Driver ioctl Operation

Purpose

Sends a Bus Device Reset (BDR) message to an attached target device.

Note: This operation is not supported by all SCSI I/O controllers.

Description

The **TMIORESET** ioctl operation allows the caller to send a Bus Device Reset (BDR) message to a selected target device. Only an initiator-mode device may use this operation. The *arg* parameter of the **TMIORESET** operation is set to a null value by the caller.

The attached target device typically uses this BDR message to reset certain operating characteristics. Such an action may be needed during severe error recovery between the host initiator and the attached target device. The specific effects of the BDR message are device dependent. Since the effects of this operation are potentially adverse to the target device, care should be taken by the caller before issuing this message. To run this operation requires at least **dev_config** authority.

Files

/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

TMIOSTAT (Status) tmscsi Device Driver ioctl Operation

Purpose

Allows the caller to get detailed status about the previous **write** or **TMGETSENS** operation.

Note: This operation is not supported by all SCSI I/O controllers.

Description

The **TMIOSTAT** operation allows the caller to get detailed status about a previous **write** or **TMGETSENS** operation. This operation is allowed only for the initiator-mode device. The *arg* parameter to this operation specifies the address of the **tm_get_stat** structure defined in **/usr/include/sys/tmscsi.h** file. The status returned by the **TMIOSTAT** operation is updated for both successful and unsuccessful completions of these commands. This status is not valid for all **errno** global variables.

Files

/dev/tmscsi0,/dev/tmscsi1,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related Information

The **rmt** SCSI device driver, **scdisk** SCSI device driver, SCSI Adapter device driver, **tmscsi** SCSI device driver.

Chapter 6. Integrated Device Electronics (IDE)

IDE Adapter Device Driver

Purpose

Supports the Integrated Device Electronics (IDE) adapter.

Syntax

```
#include </usr/include/sys/ide.h>
#include </usr/include/sys/devinfo.h>
```

Description

The **/dev/ide***n* special files provide interfaces to allow IDE device drivers to access IDE devices. These files manage the adapter resources so that multiple IDE device drivers can access devices on the same IDE adapter simultaneously. IDE adapters are accessed through the special files **/dev/ide0**, **/dev/ide1**, and similarly named files.

The **/dev/ide***n* special files provide interfaces for access to the IDE adapter. The host adapter is an initiator for access to devices such as disks, tapes, and CD-ROMs.

Device-Dependent Subroutines

The IDE adapter device driver supports only the **open**, **close**, and **ioctl** subroutines. The **read** and **write** subroutines are not supported.

open and close Subroutines

> **Note:** The IDE Adapter device driver's **open** and **close** subroutines do not support diagnostic open. If such an open is attempted, the subroutine returns a value of -1 and the **errno** global value is set to **EINVAL**.

Any kernel process can open the IDE adapter device driver in normal mode. For normal mode the *ext* parameter is set to 0. However, a non-kernel process must have at least **dev_config** authority to open the IDE adapter device driver in normal mode. Attempting to execute a normal **open** subroutine without the proper authority causes the subroutine to return a value of -1, and set the **errno** global variable to a value of **EPERM**.

ioctl Subroutine

Along with the **IOCINFO** operation, the IDE device driver defines specific operations for devices.

The **IOCINFO** operation is defined for all device drivers that use the **ioctl** subroutine, as follows:

- The operation returns a **devinfo** structure. This structure is defined in the **/usr/include/sys/devinfo.h** file. The device type in this structure is **DD_BUS**, and the subtype is **DS_IDE**. The flags field is not used and is set to 0.
- The **devinfo** structure includes unique data such as the maximum data transfer size allowed (in bytes). A calling IDE device driver uses this information to learn the maximum transfer size allowed for a device it controls on the IDE adapter. In this way, the IDE device driver can control devices across various IDE adapters, with each device possibly having a different maximum transfer size.

IDE ioctl Operations for Adapters

The operations are IDE adapter device driver functions, rather than general device driver facilities.

The following IDE operations are for adapters:

Operation	Description
IDEIOGTHW	Allows the caller to verify IDE adapter device driver support for gathered writes.
IDEIOINQU	Provides the means to issue an inquire command to an ATAPI IDE device.
IDEIOIDENT	Provides the means to issue an identify device command to an IDE device. An indicator is returned which identifies if the device is an ATA or ATAPI type device.
IDEIOREAD	Sends a single block **read** command to the selected ATA IDE device, this is not supported for ATAPI type devices.
IDEIORESET	Allows the caller to force an IDE device to clear all current commands and return to an initial state.
IDEIOSTART	Opens a logical path to an IDE target device.
IDEIOSTOP	Closes the logical path to an IDE target device.
IDEIOSTUNIT	Provides the means to issue an IDE Start Unit command to a selected ATAPI IDE device.
IDEIOTUR	Sends a Test Unit Ready command to the selected ATAPI IDE device.

Summary of IDE Error Conditions

Possible errno values for the adapter device driver are:

Value	Description
EACCES	Indicates that an **openx** subroutine was attempted while the adapter had one or more devices in use.
EBUSY	Indicates that a delete operation was unsuccessful. The adapter is still open.
EFAULT	Indicates that a copy between kernel and user space failed.
EINVAL	Indicates an invalid parameter or that the device has not been opened.
EIO	Indicates an invalid command. A **IDEIOSTART** operation must be executed prior to this command, or an invalid IDE master or slave was passed in.
EIO	Indicates that the command has failed due to an error detected on the adapter or the IDE bus.
EIO	Indicates that the device driver was unable to pin code.
EIO	Indicates that a kernel service failed, or that an unrecoverable I/O error occurred.
ENOCONNECT	Indicates that an IDE bus fault occurred.
ENODEV	Indicates that the target device cannot be selected or is not responding.
ENOMEM	Indicates that the command could not be completed due to an insufficient amount of memory.
ENXIO	Indicates that the requested ioctl is not supported by this adapter.
EPERM	Indicates that the caller did not have the required authority.
ETIMEDOUT	Indicates that an IDE command has exceeded the time-out value.

Reliability and Serviceability Information

Errors detected by the adapter device driver may be one of the following:

- Permanent adapter or system hardware errors
- Temporary DMA error
- Temporary unknown adapter device driver errors
- Temporary error for command timeout

Permanent errors are either errors that cannot be retried or errors not recovered before a prescribed number of retries has been exhausted. Temporary errors are either noncatastrophic errors that cannot be retried or retriable errors that are successfully recovered before a prescribed number of retries has been exhausted.

Error Record Values for Permanent Hardware Errors

The error record template for permanent hardware errors detected by the IDE adapter device driver is described below. Refer to the **ataide_rc** structure for the actual definition of the detail data. The **ataide_rc** structure is defined in the **/usr/include/sys/ide.h** file:

Field	Description
Comment	Indicates ATA/IDE controller reset failure.
Class	Equals a value of H, which indicates a hardware error.
Report	Equals a value of TRUE, which indicates this error should be included when an error report is generated.
Log	Equals a value of TRUE, which indicates an error log entry should be created when this error occurs.
Alert	Equals a value of FALSE, which indicates this error is not alertable.
Err_Type	Equals a value of PERM, which indicates a permanent failure.
Err_Desc	Equals a value of 0xE98A, which indicates an adapter reset failure.
Prob_Causes	The following: • 0xE901, which indicates a configuration error • 0x3452, which indicates a storage device cable • 0x6310, which indicates a DASD device • 0xEA01, which indicates an adapter failure
Fail_Causes	The following: • 0x3400, which indicates a cable loose or defective • 0x3303, which indicates a DASD adapter
Fail_Actions	The following: • 0x0301, which indicates to check the cables and its connections. • 0x0000, which indicates to perform a problem determination procedure.
Detail_Data1	Equals a value of 56, EC35, and HEX

The error record template for DMA errors detected by the IDE adapter device driver follows:

Field	Description
Comment	Indicates IDE DMA transfer error.
Class	Equals a value of H, which indicates a hardware error.
Report	Equals a value of TRUE, which indicates this error should be included when an error report is generated.
Log	Equals a value of TRUE, which indicates an error-log entry should be created when this error occurs.
Alert	Equals a value of FALSE, which indicates this error is not alertable.
Err_Type	Equals a value of TEMP, which indicates a temporary failure.
Err_Desc	Equals a value of 0xEB75, which indicates DMA error.
Prob_Causes	The following: • 0xE901, which indicates a configuration error • 0x3452, which indicates a storage device cable • 0x6310, which indicates a DASD device • 0xEA01, which indicates an adapter failure
Fail_Causes	The following: • 0x3400, which indicates a cable loose or defective • 0x3303, which indicates a DASD adapter
Fail_Actions	The following: • 0x0301, which indicates to check the cable and its connections. • 0x0000, which indicates to perform problem-determination procedure.
Detail_Data1	Equals a value of 56, EC35, HEX

Error Record Values for Temporary Unknown IDE Device Errors

The error-record template for unknown IDE adapter errors detected by the IDE adapter device driver follows:

Field	Description
Comment	Indicates IDE Device error
Class	Equals a value of H, which indicates a hardware error.
Report	Equals a value of TRUE, which indicates this error should be included when an error report is generated.
Log	Equals a value of TRUE, which indicates an error log entry should be created when this error occurs.
Alert	Equals a value of FALSE, which indicates this error is not alertable.
Err_Type	Equals a value of TEMP, which indicates a temporary failure.
Err_Desc	Equals a value of 0x1002, which indicates device error.
Prob_Causes	The following: • 0xE901, which indicates configuration error • 0x3452, which indicates storage device cable • 0x6310, which indicates DASD device • 0xEA03, which indicates adapter error
Fail_Causes	The following: • 0x3400, which indicates a cable loose or defective • 0x3303, which indicates DASD adapter
Fail_Actions	The following: • 0x0301, which indicates to check the cable and its connections. • 0x0000, which indicates to perform problem-determination procedure.
Detail_Data1	Equals a value of 56, EC35, HEX.

Error Record Values for IDE Command Timeout Errors

The error-record template for IDE Command Timeout errors detected by the IDE adapter device driver follows:

Field	Description
Comment	Indicates IDE Interrupt timeout error.
Class	Equals a value of H, which indicates a hardware error.
Report	Equals a value of TRUE, which indicates this error should be included when an error report is generated.
Log	Equals a value of TRUE, which indicates an error log entry should be created when this error occurs.
Alert	Equals a value of FALSE, which indicates this error is not alertable.
Err_Type	Equals a value of TEMP, which indicates a temporary failure.
Err_Desc	Equals a value of 0xE96B, which indicates interrupt timed out.
Prob_Causes	The following: • 0xE901, which indicates a configuration error • 0x3452, which indicates a storage device cable • 0x6310, which indicates a DASD device • 0xEA01, which indicates an adapter failure
Fail_Causes	The following: • 0x3400, which indicates a cable loose or defective • 0x3303, which indicates DASD adapter
Fail_Actions	The following: • 0x0301, which indicates to check the cable and its connections. • 0x0000, which indicates to perform problem-determination procedures.
Detail_Data1	Equals a value of 56, EC35, HEX.

Managing Dumps

The IDE adapter device driver is a target for the system dump facility. The **DUMPINIT** and **DUMPSTART** options to the **dddump** entry point support multiple or redundant calls. The **DUMPQUERY** option returns a minimum transfer size of 0 bytes and a maximum transfer size equal to the maximum transfer size supported by the IDE adapter device driver.

To be processed, calls to the IDE adapter device driver **DUMPWRITE** option should use the *arg* parameter as a pointer to the **ataide_buf** structure. Using this interface, a IDE write command can be run on a previously started (opened) target device. The *uiop* parameter is ignored by the IDE adapter device driver. Spanned or consolidated commands are not supported using **DUMPWRITE**.

> **Note:** The various **ataide_buf** status fields, including the **b_error** field, are not set at completion of the **DUMPWRITE**. Error logging is, of necessity, not supported during the dump.

Successful completion of the **dddump** entry point is indicated by a 0. If unsuccessful, the entry point returns one of the following:

Return	Description
EINVAL	Indicates that the adapter device driver was passed as a request that was invalid, such as attempting a **DUMPSTART** option before successfully executing a **DUMPINIT** option.
EIO	Indicates that the adapter device driver was unable to complete the command due to a lack of required resources or an I/O error.
ETIMEDOUT	Indicates that the adapter did not respond with status before the passed command time-out value expired.

Special Files

/dev/ide0, /dev/ide1,..., **/dev/ide***n*	Provides an interface to allow IDE device drivers to access IDE devices or adapters.

Related Information

idedisk IDE device driver or idecdrom IDE device driver.

---

idecdrom IDE Device Driver

Purpose

Supports the Integrate Device Electronics (IDE) CD-ROM devices.

Syntax

```
#include <sys/devinfo.h>
#include <sys/ide.h>
#include <sys/idecdrom.h>
```

Device-Dependent Subroutines

Typical CD-ROM drive operations are implemented using the **open**, **close**, **read**, and **ioctl** subroutines.

open and close Subroutines

The **openx** subroutine is intended primarily for use by the utilities.

The *ext* parameter passed to the **openx** subroutine selects the operation to be used for the target device. The **/usr/include/sys/idecdrom.h** file defines possible values for the *ext* parameter.

The *ext* parameter can contain any combination of the following flag values logically ORed together:

Flag Value	Description
IDE_SINGLE	Places the selected device in exclusive access mode. Only one process at a time can open a device in exclusive access mode.

A device can be opened in exclusive access mode only if the device is not currently open. If an attempt is made to open a device in exclusive access mode and the device is already open, the subroutine returns a value of -1 and sets the **errno** global variable to a value of EBUSY.

ioctl Subroutine

ioctl subroutine operations that are used for the idecdrom device driver are:

Operation	Description
IOCINFO	Returns the **devinfo** structure defined in the **/usr/include/sys/devinfo**.h file. The **IOCINFO** operation is the only operation defined for all device drivers that use the **ioctl** subroutine. The remaining operations discussed in this article are all specific to CD-ROM devices.
IDE_CDIORDSE	Provides a means for issuing a **read** command to the device and obtaining the target-device sense data when an error occurs. If the **IDE_CDIORDSE** operation returns a value of -1 and the **status_validity** field has the **ATA_ERROR** bit set, valid sense data is returned. Otherwise, target sense data is omitted.

The **IDE_CDIORDSE** operation is provided for diagnostic use. It allows the limited use of the target device while operating in an active system environment. The *arg* parameter to the **IDE_CDIOROSE** operation contains the address of an **sc_rdwrt** structure. This structure is defined in the **/usr/include/sys/scsi.h** file.

The **devinfo** structure defines the maximum transfer size for a read operation. If an attempt is made to transfer more than the maximum, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. Refer to the ATA Packet Interface for CD-ROMS Specification for the format of the request-sense data for a particular device.

Note: The **IDE_CDIORDSE** operation can be substituted for the **DKIORDSE** operation when issuing a **read** command to obtain sense data from a CD-ROM device.

Operation	Description
IDE_CDPMR	Issues an IDE prevent media removal command when the device has been successfully opened. This command prevents media from being ejected until the device is closed, powered off and then back on, or until a **IDE_CDAMR** operation is issued. The *arg* parameter for the **IDE_CDPMR** operation is null. If the **IDE_CDPMR** operation is successful, the subroutine returns a value of 0. If the **IDE_CDPMR** operation fails for any reason, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EIO**.
IDE_CDAMR	Issues an allow media removal command when the device has been successfully opened. As a result media can be ejected using either the drive's eject button or the **IDE_CDEJECT** operation. The *arg* parameter for this ioctl is null. If the **IDE_CDAMR** operation is successful, the subroutine returns a value of 0. For any failure of this operation, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EIO**.
IDE_CDEJECT	Issues an eject media command to the drive when the device has been successfully opened. The *arg* parameter for this operation is null. If the **IDE_CDEJECT** operation is successful, the subroutine returns a value of 0. For any failure of this operation, the subroutine returns a value of -1 and sets the **errno** variable to a value of **EIO**.

Operation	Description
IDE_CDAUDIO	Issues play-audio commands to the specified device and controls the volume on the device's output ports. Play audio commands include: play audio MSF (play audio track index is not supported), pause, resume, stop, determine the number of tracks, and determine the status of a current audio operation. The **IDE_CDAUDIO** operation plays audio only through the CD-ROM drive's output ports. The *arg* parameter of this operation is the address of a **cd_audio_cmd** structure, which is defined in the **/usr/include/sys/scdisk.h** file. Exclusive access mode is required. If **IDE_CDAUDIO** operation is attempted when the device's audio-supported attribute is set to no, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. If the **IDE_CDAUDIO** operation fails, the subroutine returns a value of -1 and sets the **errno** global variable to a nonzero value. In this case, the caller should evaluate the returned status bytes to determine why the operation failed and what recovery actions should be taken.
IDE_CDMODE	Determines or changes the CD-ROM data mode for the specified device. The CD-ROM data mode specifies what block size and special file are used for data read across the IDE bus from the device.The **IDE_CDMODE** operation supports the following CD-ROM data modes: **CD_ROM Data Mode 1** 2048-byte block size through both raw (**dev/rcd***) and block special (**/dev/cd***) files **CD_ROM Data Mode 2 Form 1** 2048-byte block size through both raw (**dev/rcd***) and block special (**/dev/cd***) files **CD_ROM Data Mode 2 Form 2** 2336-byte block size through the raw (**dev/rcd***) special file only **CD_DA (Compact Disc Digital Audio)** 2352-byte block size through the raw (**dev/rcd***) special file only The **IDE_CDMODE** *arg* parameter contains the address of the **mode_form_op** structure defined in the **/usr/include/sys/scdisk.h** file. To have the **IDE_CDMODE** operation determine or change the CD-ROM data mode, set the action field of the **change_mode_form** structure to one of the following values: **CD_GET_MODE** Returns the current CD-ROM data mode in the `cd_mode_form` field of the **mode_form_op** structure, when the device has been successfully opened. **CD_CHG_MODE** Changes the CD-ROM data mode to the mode specified in the `cd_mode_form` field of the **mode_form_op** structure, when the device has been successfully opened in the exclusive access mode.

If a CD-ROM has not been configured for different data modes, and an attempt is made to change the CD-ROM data mode (by setting the action field of the **change_mode_form** structure set to **CD_CHG_MODE**), the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**.

If the **IDE_CDMODE** operation for **CD_CHG_MODE** is attempted when the device is not in exclusive access mode, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EACCES**. For any other failure of this operation, the subroutine returns a value of -1 and sets the **errno** global variable to a value of EIO.

Device Requirements

IDE CD-ROM drives have the following hardware requirements:

- IDE CD-ROM drive must support the IDE **ATAPI_READ_CD** command.
- If a IDE CD-ROM drive uses **CD_ROM** Data Mode 1, it must support a block size of 2048 bytes per block.

- If an IDE CD-ROM drive uses **CD_ROM** Data Mode 2 Form 1, it must support a block size of 2048 bytes per block.
- If an IDE CD-ROM drive uses **CD_ROM** Data Mode 2 Form 2, it must support a block size of 2336 bytes per block.
- If an IDE CD-ROM drive uses **CD_DA** mode, it must support a block size of 2352 bytes per block.
- To control volume using the **IDE_CDAUDIO** (play-audio) operation, the device must support mode data page 0xE.
- To use the **IDE_CDAUDIO** (play-audio) operation, the device must support the following optional commands:
  - read sub-channel
  - pause resume
  - play audio MSF
  - read TOC

Error Conditions

Possible **errno** values for **ioctl**, **open**, **read**, and **write** subroutines when using the idecdrom device driver include:

Value	Description
EACCES	Indicates one of the following circumstances: • An attempt was made to open a device currently open exclusive access mode. • An **IDE_CDMODE ioctl** subroutine operation was attempted on a device not in exclusive access mode.
EBUSY	An attempt was made to open a session in exclusive access mode on a device already opened.
EFAULT	Indicates an illegal user address.
EFORMAT	Indicates the target device has unformatted media or media in an incompatible format.
EINPROGRESS	Indicates a CD-ROM drive has a play-audio operation in progress.
EINVAL	Indicates one of the following circumstances: • An **IDE_CDAUDIO** (play-audio) operation was attempted for a device that is not configured to use the IDE play-audio commands. • The read subroutine supplied an *nbyte* parameter that is not an even multiple of the block size. • A sense data buffer length of greater than 255 bytes is not valid for a **IDE_CDIORDSE ioctl** subroutine operation. • The data buffer length exceeded the maximum defined in the **devinfo** structure for a **IDE_CDIORDSE ioctl** subroutine operation. • An unsupported **ioctl** subroutine operation was attempted. • An attempt was made to configure a device that is still open. • An illegal configuration command has been given. • An **IDE_CDPMR** (Prevent Media Removal), **IDE_CDAMR** (Allow Media Removal), or **IDE_CDEJECT** (Eject Media) command was sent to a device that does not support removable media. • An **IDE_CDEJECT** (Eject Media) command was sent to a device that currently has its media locked in the drive. • The data buffer length exceeded the maximum defined for a strategy operation.
EIO	Indicates one of the following circumstances: • The target device cannot be located or is not responding. • The target device has indicated an unrecovered hardware error.

Value	Description
EMEDIA	Indicates one of the following circumstances: • The target device has indicated an unrecovered media error. • The media was changed.
EMFILE	Indicates an **open** operation was attempted for an adapter that already has the maximum permissible number of opened devices.
ENODEV	Indicates one of the following circumstances: • An attempt was made to access an undefined device. • An attempt was made to close an undefined device.
ENOTREADY	Indicates no media is in the drive.
ENXIO	Indicates one of the following circumstances: • The **ioctl** subroutine supplied an invalid parameter.
EPERM	Indicates the attempted subroutine requires appropriate authority.
ESTALE	Indicates a read-only disk was ejected (without first being closed by the user) and then either reinserted or replaced with a second disk.
ETIMEDOUT	Indicates an **I/O** operation has exceeded the given timer value.
EWRPROTECT	Indicates one of the following circumstances: • An **open** operation requesting read/write mode was attempted on read-only media. • A **write** operation was attempted to read-only media.

Reliability and Serviceability Information

IDE CD-ROM drives return the following errors:

Error	Description
ABORTED COMMAND	Indicates the device ended the command.
GOOD COMPLETION	Indicates the command completed successfully.
HARDWARE ERROR	Indicates an unrecoverable hardware failure occurred during command execution or during a self-test.
ILLEGAL REQUEST	Indicates an illegal command or command parameter.
MEDIUM ERROR	Indicates the command ended with an unrecovered media error condition.
NOT READY	Indicates the logical unit is offline or media is missing.
RECOVERED ERROR	Indicates the command was successful after some recovery was applied.
UNIT ATTENTION	Indicates the device has been reset or the power has been turned on.

Error Record Values for Media Errors

The fields defined in the error record template for CD-ROM media errors are:

Field	Description
`Comment`	Indicates CD-ROM read media error.
`Class`	Equals a value of H, which indicates a hardware error.
`Report`	Equals a value of True, which indicates this error should be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error log entry should be created when this error occurs.
`Alert`	Equals a value of False, which indicates this error is not alertable.
`Err_Type`	Equals a value of Perm, which indicates a permanent failure.
`Err_Desc`	Equals a value of 1312, which indicates a disk operation failure.
`Prob_Causes`	Equals a value of 5000, which indicates media.
`User_Causes`	Equals a value of 5100, which indicates the media is defective.
`User_Actions`	Equals the following values: • 0000, which indicates problem-determination procedures should be performed • 1601, which indicates the removable media should be replaced and retried

Field	Description
Fail_Causes	Equals the following values: • 5000, which indicates a media failure • 6310, which indicates a disk drive failure
Fail_Actions	Equals the following values: • 0000, which indicates problem-determination procedures should be performed • 1601, which indicates the removable media should be replaced and retried
Detail_Data	Equals a value of 156, 11, HEX. This value indicates hexadecimal format.

Note: The Detail_Data field in the **err_rec** structure contains the **idecdrom_error_rec** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **idecdrom_error_rec** structure is defined in the **/usr/include/sys/ide.h** file.

The **idecdrom_error_rec** structure contains the following fields:

Field	Description
req_sense_data	Contains the request-sense information from the particular device that had the error, if it is valid.
reserved2	Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.
reserved3	Contains the number of bytes read since the segment count was last increased.

Refer to the ATA Packet Interface for CD-ROMs Specification for the format of the request-sense data for a particular device.

Error Record Values for Hardware Errors

The fields defined in the error record template for CD-ROM hardware errors, as well as hard-aborted command errors are:

Field	Description
Comment	Indicates CD-ROM hardware error.
Class	Equals a value of H, which indicates a hardware error.
Report	Equals a value of True, which indicates this error should be included when an error report is generated.
Log	Equals a value of True, which indicates an error log entry should be created when this error occurs.
Alert	Equal to a value of FALSE, which indicates this error is not alertable.
Err_Type	Equals a value of Perm, which indicates a permanent failure.
Err_Desc	Equals a value of 1312, which indicates a disk operation failure.
Prob_Causes	Equals a value of 6310, which indicates disk drive.
Fail_Causes	Equals the following values: • 6310, which indicates a disk drive failure • 6330, which indicates a disk drive electronics failure
Fail_Actions	Equals a value of 0000, which indicates problem-determination procedures should be performed.
Detail_Data	Equals a value of 156, 11, HEX. This value indicates hexadecimal format.

Note: he Detail_Data field in the **err_rec** structure contains the **idecdrom_error_rec** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **idecdrom_error_rec** structure is defined in the **/usr/include/sys/ide.h** file.

The **idecdrom_error_rec** structure contains the following fields:

Field	Description
req_sense_data	Contains the request-sense information from the particular device that had the error, if it is valid.
reserved2	Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.
reserved3	Contains the number of bytes read since the segment count was last increased.

Refer to the ATA Packet Interface for CD-ROMs Specification for the format of the request-sense data for a particular device.

Error Record Values for Recovered Errors

The fields defined in the error record template for CD-ROM media errors recovered errors are:

Field	Description
`Comment`	Indicates CD-ROM recovered error.
`Class`	Equals a value of H, which indicates a hardware error.
`Report`	Equals a value of True, which indicates this error should be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error log entry should be created when this error occurs.
`Alert`	Equal to a value of FALSE, which indicates this error is not alertable.
`Err_Type`	Equals a value of Temp, which indicates a temporary failure.
`Err_Desc`	Equals a value of 1312, which indicates a physical volume operation failure.
`Prob_Causes`	Equals the following values: • 5000, which indicates a media failure • 6310, which indicates a disk drive failure
`User_Causes`	Equals a value of 5100, which indicates media is defective.
`User_Actions`	Equals the following values: • 0000, which indicates problem-determination procedures should be performed • 1601, which indicates the removable media should be replaced and retried
`Fail_Causes`	Equals the following values: • 5000, which indicates a media failure • 6310, which indicates a disk drive failure
`Fail_Actions`	Equals the following values: • 0000, which indicates problem-determination procedures should be performed • 1601, which indicates the removable media should be replaced and retried
`Detail_Data`	Equals a value of 156, 11, HEX. This value indicates hexadecimal format.

Note: The `Detail_Data` field in the **err_rec** structure contains the **idecdrom_error_rec** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **idecdrom_error_rec** structure is defined in the **/usr/include/sys/ide.h** file.

The **idecdrom_error_rec** structure contains the following fields:

Field	Description
req_sense_data	Contains the request-sense information from the particular device that had the error, if it is valid.
reserved2	Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.
reserved3	Contains the number of bytes read since the segment count was last increased.

Refer to the ATA Packet Interface for CD-ROMs Specification for the format of the request-sense data for a particular device.

Error Record Values for Unknown Errors

The fields defined in the error record template for CD-ROM media errors unknown errors are:

Field	Description
Comment	Indicates CD-ROM unknown failure.
Class	Equals a value of H, which indicates a hardware error.
Report	Equals a value of True, which indicates this error should be included when an error report is generated.
Log	Equals a value of True, which indicates an error log entry should be created when this error occurs.
Alert	Equal to a value of FALSE, which indicates this error is not alertable.
Err_Type	Equals a value of Unkn, which indicates the type of error is unknown.
Err_Desc	Equals a value of FE00, which indicates an undetermined error.
Prob_Causes	Equals the following values: • 3300, which indicates an adapter failure • 5000, which indicates a media failure • 6310, which indicates a disk drive failure
Fail_Causes	Equals a value of FFFF, which indicates the failure causes are unknown.
Fail_Actions	Equals the following values: • 0000, which indicates problem-determination procedures should be performed • 1601, which indicates the removable media should be replaced and retried
Detail_Data	Equals a value of 156, 11, HEX. This value indicates hexadecimal format.

Note: The Detail_Data field in the **err_rec** structure contains the **idecdrom_error_rec** structure. The err_rec structure is defined in the **/usr/include/sys/errids.h** file. The **idecdrom_error_rec** structure is defined in the **/usr/include/sys/ide.h** file.

The **idecdrom_error_rec** structure contains the following fields:

Field	Description
req_sense_data	Contains the request-sense information from the particular device that had the error, if it is valid.
reserved2	Contains the segment count, which is the number of megabytes read from the device at the time the error occurred.
reserved3	Contains the number of bytes read since the segment count was last increased.

Refer to the ATA Packet Interface for CD-ROMs Specification for the format of the request-sense data for a particular device.

Special Files

The idecdrom IDE device driver uses raw and block special files in performing its functions.

Attention: Data corruption, loss of data, or loss of system integrity (system crash) will occur if devices supporting paging, logical volumes, or mounted file systems are accessed using block special files. Block special files are provided for logical volumes and disk devices and are solely for system use in managing file systems, paging devices, and logical volumes. These files should not be used for other purposes.

The special files used by the idecdrom, device driver include the following:

File	Description
/dev/rcd0, /dev/rcd1,..., /dev/rcd*n*	Provide an interface to allow IDE device drivers character access (raw I/O access and control functions) to IDE CD-ROM disks.
/dev/cd0, /dev/cd1,..., /dev/cd*n*	Provide an interface to allow IDE device drivers block I/O access to IDE CD-ROM disks.

The prefix **r** on a special file name indicates the drive is accessed as a raw device rather than a block device. Performing raw I/O with a CD-ROM drive requires that all data transfers be in multiples of the device block size. Also, all **lseek** subroutines that are made to the raw device driver must result in a file pointer value that is a multiple of the device block size.

Related Information

"IDE Adapter Device Driver" on page 335.

Special Files, and cd Special File in *AIX 5L Version 5.2 Files Reference*.

Integrated Device Electronics (IDE) Subsystem, A Typical IDE Driver Transaction Sequence, Required IDE Adapter Device Driver ioctl Commands, Understanding the Execution of Initiator I/O Requests, IDE Error Recovery, and ataide_buf Structure in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

close Subroutine, ioctl, ioctlx, ioctl32, or ioctl32x Subroutine, and open, openx, open64, creat, or creat64 Subroutine in *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1*.

read, readx, readv, readvx, or pread Subroutine, and write, writex, writev, writevx or pwrite Subroutines in *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2*

---

idedisk IDE Device Driver

Purpose

Supports the Integrated Device Electronics (IDE) fixed disk devices.

Syntax

```
#include <sys/devinfo.h>
#include <sys/ide.h>
```

Device-Dependent Subroutines

Typical fixed disk operations are implemented using the **open**, **close**, **read**, **write**, and **ioctl** subroutines.

open and close Subroutines

The standard **open** and **close** operations are supported by the idedisk device driver. The openx operation is not supported.

readx and writex Subroutines

The **readx** and **writex** subroutines provide additional parameters affecting the raw data transfer. These subroutines pass the *ext* parameter, which specifies request options. The options are constructed by logically ORing zero or more of the following values:

WRITEV Indicates a request for write verification.

ioctl Subroutine

ioctl subroutine operations that are used for the idedisk device driver are:

Operation	Description
IOCINFO	Returns the **devinfo** structure defined in the **/usr/include/sys/devinfo.h** file. The **IOCINFO** operation is the only operation defined for all device drivers that use the **ioctl** subroutine. The remaining operations discussed in this article are all specific to IDE fixed disk devices.
DKFORMAT	The IDE disk device driver does not support low level formatting of an IDE disk. IDE disks are preformatted at the factory and should not be reformatted. Attempting to format an IDE disk will result in a -1 return code and errno set to **EINVAL**.

Device Requirements

IDE fixed disks must support the following ATA commands. The commands' hexadecimal opcodes are specified in the parenthesis:

- Identify Device (EC)
- Set Features (EF)
- Initialize Drive Parameters (91)
- Read Sector(s) without retry (21)
- Read Sector(s) with retry (20)
- Write Sector(s) with retry (30)
- Read DMA with retry (C8)
- Write DMA with retry (CA)
- Read Verify with retry (40)
- Idle Immediate (95)
- Standby Immediate (94)

Error Conditions

Possible **errno** values for **ioctl**, **open**, **read**, and **write** subroutines when using the idedisk device driver include:

Value	Description
EFAULT	Indicates an illegal user address.
EINVAL	Indicates one of the following circumstances: • The **read** or **write** subroutine supplied an *nbyte* parameter that is not an even multiple of the block size. • An unsupported **ioctl** subroutine operation was attempted. • An attempt was made to configure a device that is still open. • An illegal configuration command was requested. • The data buffer length exceeded the maximum defined for a strategy operation. • The HWRELOC or UNSAFEREL bits of the **writex** *ext* parameter were set.
EIO	Indicates one of the following circumstances: • The target device cannot be located or is not responding. • The target device has indicated an unrecovered hardware error.
ENFILE	Indicates the system file table was full.
ENODEV	The specified fixed disk was not configured or does not exist.
ENXIO	Indicates one of the following circumstances: • The specified fixed disk was not opened. • The **ioctl** subroutine supplied an invalid parameter. • A **read** or **write** operation was attempted beyond the end of the fixed disk.
EPERM	Indicates the attempted subroutine requires appropriate authority.

Value	Description
ETIMEDOUT	Indicates an **I/O** operation exceeded the given time-out value.
EWRPROTECT	Indicates one of the following circumstances: • An **open** operation requesting read/write mode was attempted on read-only media. • A **write** operation was attempted to read-only media.

Reliability and Serviceability Information

The following classes of errors are reported by the IDE disk device driver:

Class	Description
SOFTWARE ACCESS ERROR	Indicates that the fixed disk was not ready to receive a command or that an unsupported command was requested.
HARDWARE ERROR	Indicates an unrecoverable hardware failure occurred during command execution.
MEDIA ERROR	Indicates an unrecoverable media error was encountered during command execution. This class of error includes bad blocks, missing sector IDs, missing address marks, recalibration failures, and uncorrectable data errors.
RECOVERED ERROR	Indicates the command succeeded due to fixed disk or disk device driver retries.

Error Record Values for Media Errors

The fields defined in the error record template for fixed disk media errors are:

Field	Description
`Comment`	Indicates fixed disk media error.
`Class`	Equals a value of H, which indicates a hardware error.
`Report`	Equals a value of True, which indicates this error should be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error log entry should be created when this error occurs.
`Alert`	Equals a value of False, which indicates this error is not alertable.
`Err_Type`	Equals a value of Perm, which indicates a permanent failure.
`Err_Desc`	Equals a value of 1312, which indicates a disk operation failure.
`Prob_Causes`	Equals the following values: • E855, which indicates a disk problem • 6330, which indicates a disk drive electronics
`Fail_Causes`	Equals the following values: • EA77, which indicates a bad block detected • EA78, which indicates an uncorrectable data error • EA79, which indicates a requested secotr's id or address mark not found • EA7A, which indicates a track 0 not found
`Fail_Actions`	Equals the following values: • EC1B, which indicates verify disk's master and slave jumpers are properly set • 0301, which indicates check cables ant its connections • 0000, which indicates problem-determination procedures should be performed
`Detail_Data`	Equals a value of 72, EC35, HEX. This value indicates hexadecimal format.

Note: The `Detail_Data` field in the **err_rec** structure contains the **idedisk_error_rec** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **idedisk_error_rec** structure is defined in the **/usr/include/sys/ide.h** file.

The **idedisk_error_rec** structure contains the following fields:

Field	Description
status_validity	Contains bit flags indicating validity of status and error fields.
b_error	Contains error value from **buf** structure.
b_flags	Contains flag value from **buf** structure.
b_addr	Contains buffer address from **buf** structure.
b_resid	Contains residual byte count from **buf** structure.
ata	Contains the IDE command that was sent to the IDE device. Also may contain command completion status.

Error Record Values for Physical Volume Software Access

The fields defined in the error record template for fixed disk physical volume software access are:

Field	Description
`Comment`	Indicates fixed disk encountered a physical volume software error.
`Class`	Equals a value of S, which indicates a software error.
`Report`	Equals a value of True, which indicates this error should be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error log entry should be created when this error occurs.
`Alert`	Equal to a value of FALSE, which indicates this error is notIDe alertable.
`Err_Type`	Equals a value of Perm, which indicates a permanent failure.
`Err_Desc`	Equals a value of 210F, which indicates a software error
`Prob_Causes`	Equals the following: • EA00, which indicates a software error • 6330, which indicates a disk drive electronics failure
`Fail_Causes`	Equals the following values: • 109B, which indicates an invalid memory request size • EA7B, which indicates an ATA status error
`Fail_Actions`	Equals the following values: • EC1B, which indicates verify disk's master and slave jumpers are properly set • 0301, which indicates check cables and its connections • 0000, which indicates problem-determination procedures should be performed
`Detail_Data`	Equals a value of 72, EC35, HEX. This value indicates hexadecimal format.

Note: The `Detail_Data` field in the **err_rec** structure contains the **idedisk_error_rec** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **idedisk_error_rec** structure is defined in the **/usr/include/sys/ide.h** file.

The **idedisk_error_rec** structure contains the following fields:

Field	Description
status_validity	Contains bit flags indicating validity of status and error fields.
b_error	Contains error value from **buf** structure.
b_flags	Contains flag value from **buf** structure.
b_addr	Contains buffer address from **buf** structure.
b_resid	Contains residual byte count from **buf** structure.
ata	Contains the IDE command that was sent to the IDE device. Also may contain command completion status.

Error Record Values for Physical Volume Hardware Error

The fields defined in the error record template for fixed disk physical volume hardware errors recovered errors are:

Field	Description
Comment	Indicates fixed disk physical volume hardware error
Class	Equals a value of H, which indicates a hardware error.
Report	Equals a value of True, which indicates this error should be included when an error report is generated.
Log	Equals a value of True, which indicates an error log entry should be created when this error occurs.
Alert	Equal to a value of FALSE, which indicates this error is not alertable.
Err_Type	Equals a value of Perm, which indicates a permanent failure.
Err_Desc	Equals a value of 1311, which indicates a physical volume operation failure
Prob_Causes	Equals a value of 6330, which indicates a disk drive electronics failure
Fail_Causes	Equals a value of EA7C, which indicates an invalid media-change status
Fail_Actions	Equals the following values: • EC1B, which indicates verify disk's master and slave jumpers are properly set • 0301, which indicates check cables and its connections • 0000, which indicates problem-determination procedures should be performed • 1804, which indicates replace device
Detail_Data	Equals a value of 72, EC35, HEX. This value indicates hexadecimal format.

Note: The Detail_Data field in the **err_rec** structure contains the **idedisk_error_rec** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **idedisk_error_rec** structure is defined in the **/usr/include/sys/ide.h** file.

The **idedisk_error_rec** structure contains the following fields:

Field	Description
b_error	Contains error value from **buf** structure.
b_flags	Contains flag value from **buf** structure.
b_addr	Contains buffer address from **buf** structure.
b_resid	Contains residual byte count from **buf** structure.
ata	Contains the IDE command that was sent to the IDE device. Also may contain command completion status.

Error Record Values for Physical Volume Recovered Error

The fields defined in the error record template for fixed disk physical volume recovered errors are:

Field	Description
Comment	Indicates fixed disk physical volume recovered error
Class	Equals a value of H, which indicates a hardware error.
Report	Equals a value of True, which indicates this error should be included when an error report is generated.
Log	Equals a value of True, which indicates an error log entry should be created when this error occurs.
Alert	Equal to a value of FALSE, which indicates this error is not alertable.
Err_Type	Equals a value of Temp, which indicates the type of error is temporary.
Err_Desc	Equals a value of EC64, which indicates a disk failure recovered during retry.
Prob_Causes	Equals the following values: • E855, which indicates an adapter failure • 6330, which indicates a disk drive electronics failure • EA00, which indicates a media failure

Field	Description
Fail_Causes	Equals the following values: • 5000, which indicates a media failure • EA7B, which indicates an IDE command error • EA7C, which indicates an invalid media change
Fail_Actions	Equals a value of 0700, which indicates no action necessary
Detail_Data	Equals a value of 72, EC35, HEX. This value indicates hexadecimal format.

Note: The Detail_Data field in the **err_rec** structure contains the **idedisk_error_rec** structure. The **err_rec** structure is defined in the **/usr/include/sys/errids.h** file. The **idedisk_error_rec** structure is defined in the **/usr/include/sys/ide.h** file.

The **idedisk_error_rec** structure contains the following fields:

Field	Description
b_error	Contains error value from **buf** structure.
b_flags	Contains flag value from **buf** structure.
b_addr	Contains buffer address from **buf** structure.
b_resid	Contains residual byte count from **buf** structure.
ata	Contains the IDE command that was sent to the IDE device. Also may contain command completion status.

Special Files

The idedisk IDE device driver uses raw and block special files in performing its functions.

Attention: Data corruption, loss of data, or loss of system integrity (system crash) will occur if devices supporting paging, logical volumes, or mounted file systems are accessed using block special files. Block special files are provided for logical volumes and disk devices and are solely for system use in managing file systems, paging devices, and logical volumes. These files should not be used for other purposes.

The special files used to access the idedisk device driver include these fixed disk devices:

Device	Description
/dev/rhdisk0, /dev/rhdisk1,..., /dev/rhdisk*n*	Provide an interface to allow character access (raw I/O access and control functions) to IDE fixed disks.
/dev/hdisk0, /dev/hdisk1,..., /dev/hdisk*n*	Provide an interface to allow block I/O access to IDE fixed disks.

Note: The prefix **r** on a special file name indicates the drive is accessed as a raw device rather than a block device. Performing raw I/O with a fixed disk drive requires that all data transfers be in multiples of the device block size. Also, all **lseek** subroutines that are made to the raw device driver must result in a file pointer value that is a multiple of the device block size.

Related Information

Special Files Overview in *AIX 5L Version 5.2 Files Reference*.

IDE Subsystem Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

A Typical IDE Driver Transaction Sequence in BkSym.PrgDevDrv;.

Required IDE Adapter Device Driver ioctl Commands in BkSym.PrgDevDrv;.

Understanding the Execution of Initiator I/O Requests in BkSym.PrgDevDrv;.

IDE Error Recovery in BkSym.PrgDevDrv;.

ataide_buf Structure in BkSym.PrgDevDrv;.

IDE Adapter Device Driver.

The **close** subroutine, **ioctl** or **ioctlx** subroutine, **open**, **openx**, **create** subroutine, **read**, **readx**, **readv**, or **readvx** subroutine, **write**, **writex**, **writev**, or **writevx** subroutine.

rhdisk Special File.

IDEIOIDENT (Identify Device) IDE Adapter Device Driver ioctl Operation

Purpose

Provides the means to issue an identify device command to an Integrated Device Electronics (IDE) ATA or ATAPI device.

Description

The **IDEIOIDENT** operation allows the caller to issue an IDE identify device command to a selected device. This command can be used by system management routines to aid in configuration of IDE devices.

The *arg* parameter for the **IDEIOIDENT** operation is the address of an **identify_device** structure. This structure is defined in the **/usr/include/sys/ide.h** file. The *identify_device* parameter block allows the caller to select the IDE device ID to be queried.

If successful, the returned device data can be found at the address specified by the caller in the **identify_device** structure. Successful completion occurs if a device responds at the requested IDE deivce ID. Refer to the ATA Specification or the ATA Packet Interface for CD-ROMs Specification or the ATA Packet Interface for Streaming Tapes Specification for the applicable device for the format of the returned data. The data within the **identify_deivce** structure is in little endian format; it normally will need to be byte swapped in order to correctly interpret the data. Each 16-bit word, at 16-bit offsets, will need to swap the most significant 8-bit byte with the least significant 8-bit byte.

Note: The IDE adapter device driver performs normal error-recovery procedures during execution of this command.

Return Values

When completed successfully this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to 1 of the following values:

Value	Description
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Indicates that an **IDEIOSTART** command was not issued prior to this command.
EIO	Indicates that an unrecoverable I/O error has occurred. In the case of an unrecovered error, the adapter error-status information is logged in the system error log.
ENOCONNECT	Indicates that a bus fault has occurred. Generally the IDE adapter device driver cannot determine which device caused the IDE bus fault, so this error is not logged.

Value	Description
ENODEV	Indicates that no IDE device responded to the requested IDE device ID. This return value implies that no device exists on the requested IDE device ID. Therefore, when the **ENODEV** return value is encountered, the caller can skip this IDE device ID and go on to the next IDE device ID. This condition is not necessarily an error and is not logged.
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates that the device did not respond with a status before the internal command time-out value expired.

Files

/dev/ide0, /dev/ide1, ..., /dev/ide*n*	Provide an interface to allow IDE device drivers to access IDE devices or adapters.

Related Information

idedisk IDE device driver or idecdrom IDE device driver.

IDEIOINQU (Inquiry) IDE Adapter Device Driver ioctl Operation

Purpose

Provides the means to issue an inquiry command to an Integrated Device Electronics (IDE) ATAPI device.

Description

The **IDEIOINQU** operation allows the caller to issue an IDE device inquiry command to a selected device. This command can be used by system management routines to aid in configuration of IDE devices.

The *arg* parameter for the **IDEIOINQU** operation is the address of an **ide_inquiry** structure. This structure is defined in the **/usr/include/sys/ide.h** file. The *ide_inquiry* parameter block allows the caller to select the IDE device ID to be queried.

If successful, the returned inquiry data can be found at the address specified by the caller in the **ide_inquiry** structure. Successful completion occurs if a device responds at the requested IDE device ID. Refer to the ATA Packet Interface for CD-ROMs Specification or ATA Packet Interface for Streaming Tapes Specification for the applicable device for the format of the returned data.

Note: The IDE adapter device driver performs normal error-recovery procedures during execution of this command.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to 1 of the following values:

Value	Description
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Indicates that a **IDEIOSTART** command was not issued prior to this command.
EIO	Indicates that an unrecoverable **I/O** error has occurred. If **EIO** is returned, the caller should retry the **IDEIOINQU** operation since the first command may have cleared an error condition with the device. In case of an unrecovered error, the adapter error-status information is logged in the system error log.
ENOCONNECT	Indicates that a bus fault has occurred. Generally the IDE adapter device driver cannot determine which device caused the IDE bus fault, so this error is not logged.

Value	Description
ENODEV	Indicates that no IDE device responded to the requested IDE device ID. This return value implies that no device exists on the requested IDE device ID. Therefore, when the **ENODEV** return value is encountered, the caller can skip this IDE device ID and go on to the next IDE device ID. This condition is not necessarily an error and is not logged.
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates that the device did not respond with a status before the internal command time-out value expired. On receiving the **ETIMEDOUT** return value, the caller should retry this command at least once, since the first command may have cleared an error condition with the device. This error is logged in the system error log.

Files

/dev/ide0, /dev/ide1, ..., /dev/ide*n*	Provide an interface to allow IDE device drivers to access IDE devices or adapters.

Related Information

idedisk IDE device driver or idecdrom IDE device driver.

IDEIOREAD (Read) IDE Adapter Device Driver ioctl Operation

Purpose

Issues a single block Integrated Device Electronics (IDE) read command to a selected IDE ATA device.

Description

The **IDEIOREAD** operation allows the caller to issue an IDE device **read** command to a selected device. System management routines use this command for configuring IDE devices.

The *arg* parameter of the **IDEIOREAD** operation is the address of an **ide_readblk** structure. This structure is defined in the **/usr/include/sys/ide.h** header file.

This command results in the IDE adapter device driver issuing an ATA READ SECTOR **read** command. The command is set up to read only a single block. The caller supplies:

- Target device IDE device ID
- Logical block number or cylinder-head-sector block number to be read
- Length (in bytes) of the block on the device
- Time-out value (in seconds) for the command
- Pointer to the application buffer where the returned data is to be placed
- Flags parameter

The maximum block length for this command is 512 bytes. The command will be rejected if the length is found to be larger than this value.

Note: The IDE adapter device driver performs normal error-recovery procedures during execution of this command.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to 1 of the following values:

Value	Description
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Indicates that an **IDEIOSTART** command was not issued prior to this command. If the **IDEIOSTART** command was issued, then this indicates the block length field value is too large.
EIO	Indicates that an **I/O** error has occurred. If an **EIO** value is returned, the caller should retry the **IDEIOREAD** operation since the first command may have cleared an error condition with the device. In the case of an adapter error, the system error log records the adapter error status information.
ENOCONNECT	Indicates that a bus fault has occurred. Generally, the IDE adapter device driver cannot determine which device caused the bus fault, so this error is not logged.
ENODEV	Indicates that no IDE device responded to the requested IDE device ID. This return value implies that no device exists at the specified IDE device ID. This condition is not necessarily an error and is not logged.
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates the device did not respond with status before the internal time-out value expired. The caller should retry this command at least once, since the first command may have cleared an error condition with the device. The system error log records this error.

Files

/dev/ide0, /dev/ide1,..., /dev/ide*n*	Provide an interface to allow IDE device drivers to access IDE devices or adapters.

Related Information

idedisk IDE device driver or idecdrom IDE device driver.

IDEIOSTART (Start IDE) IDE Adapter Device Driver ioctl Operation

Purpose

Opens a logical path to an Integrated Device Electronics (IDE) device.

Description

The **IDEIOSTART** operation opens a logical path to an IDE device. This operation causes the adapter device driver to allocate and initialize the data areas needed to manage commands to a particular IDE device.

The **IDEIOSTART** operation must be issued prior to any of the other operations, such as **IDEIOINQU** and **IDEIORESET**. However, the **IDEIOSTART** operation is not required prior to calling the **IOCINFO** operation. Finally, when the caller is finished issuing commands to the IDE device, the **IDEIOSTOP** operation must be issued to release allocated data areas and close the path to the device.

The *arg* parameter to **IDEIOSTART** allows the caller to specify the IDE device ID identifier of the device to be started. The least significant byte in the *arg* parameter is the IDE device ID (master=0, slave=1). The remaining bytes are reserved and must be set to a value of 0.

Return Values

If completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable set to 1 of the following values:

Value	Description
EIO	Indicates either an unrecoverable **I/O** error, or the device driver is unable to pin code.
EINVAL	Indicates that the IDE device ID was incorrect.

If the **IDEIOSTART** operation is unsuccessful, the caller must not attempt other operations to this IDE device ID, since it is either already in use or was never successfully started.

Files

/dev/ide0, /dev/ide1, ..., /dev/ide*n*	Provide an interface to allow IDE device drivers to access IDE devices or adapters.

Related Information

idedisk IDE device driver or idecdrom IDE device driver.

IDEIOSTOP (Stop) IDE Adapter Device Driver ioctl Operation

Purpose

Closes the logical path to an Integrated Device Electronics (IDE) device.

Description

The **IDEIOSTOP** operation closes the logical path to an IDE device. The **IDEIOSTOP** operation causes the adapter device driver to deallocate data areas allocated in response to an **IDEIOSTART** operation. This command must be issued when the caller wishes to cease communications to a particular IDE device. The **IDEIOSTOP** operation should only be issued for a device successfully opened by a previous call to an **IDEIOSTART** operation.

The **IDEIOSTOP** operation passes the *arg* parameter. This parameter allows the caller to specify the IDE device ID of the device to be stopped. The least significant byte in the *arg* parameter is the IDE device ID. The remaining bytes are reserved and must be set to 0.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to 1 of the following values:

Value	Description
EINVAL	Indicates that the device has not been opened. An **IDEIOSTART** operation should be issued prior to calling the **IDEIOSTOP** operation.
EIO	Indicates that the device drive was unable to pin code.

Files

/dev/ide0, /dev/ide1, ..., /dev/ide*n*	Provide an interface to allow IDE device drivers to access IDE devices or adapters.

Related Information

idedisk IDE device driver or idecdrom IDE device driver.

IDEIOSTUNIT (Start Unit) IDE Adapter Device Driver ioctl Operation

Purpose

Provides the means to issue an Integrated Device Electronics (IDE) **IDE Start Unit** command to a selected IDE ATAPI device.

Description

The **IDEIOSTUNIT** operation allows the caller to issue an **IDE Start Unit** command to a selected IDE device. This command can be used by system management routines to aid in configuration of IDE devices. For the **IDEIOSTUNIT** operation, the *arg* parameter operation is the address of an **ide_startunit** structure. This structure is defined in the **/usr/include/sys/ide.h** file.

The **ide_startunit** structure allows the caller to specify the IDE device ID of the device on the IDE adapter that is to be started.

The `start_flag` field in the parameter block allows the caller to indicate the start option to the **IDEIOSTUNIT** operation. When the `start_flag` field is set to TRUE, the logical unit is to be made ready for use. When FALSE, the logical unit is to be stopped.

> **Note:** The IDE adapter device driver performs normal error-recovery procedures during execution of the **IDEIOSTUNIT** operation.

Return Values

When completed successfully, the **IDEIOSTUNIT** operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to 1 of the following values:

Value	Description
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Indicates that an **IDEIOSTART** command was not issued prior to this command.
EIO	Indicates that an unrecoverable **I/O** error has occurred. If **EIO** is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. In case of an unrecovered error, the adapter error-status information is logged in the system error log.
ENOCONNECT	Indicates that a bus fault has occurred. Generally the IDE adapter device driver cannot determine which device caused the IDE bus fault, so this error is not logged.
ENODEV	Indicates that no IDE device responded to the requested IDE device ID. This condition is not necessarily an error and is not logged.
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates that the device did not respond with status before the internal command time-out value expired. If **ETIMEDOUT** is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. This error is logged in the system error log.

Files

/dev/ide0, /dev/ide1,..., /dev/ide*n*	Provide an interface to allow IDE device drivers to access IDE devices or adapters.

Related Information

idedisk IDE device driver or idecdrom IDE device driver.

IDEIOTUR (Test Unit Ready) IDE Adapter Device Driver ioctl Operation

Purpose

Sends a **Test Unit Ready** command to the selected Integrated Device Electronics (IDE) ATAPI device.

Description

The **IDEIOTUR** operation allows the caller to issue an **IDE Test Unit Ready** command to a selected IDE device. This command is used by system management routines to help configure IDE devices.

The **ide_ready** structure allows the caller to specify the IDE device ID of the device on the IDE adapter that is to receive the **IDEIOTUR** operation. The **ide_ready** structure provides two output fields:`status_validity` and `ata_status`. Using these two fields, the **IDEIOTUR** operation returns the status to the caller. The *arg* parameter for the **IDEIOTUR** operation specifies the address of the **ide_ready** structure, defined in the **/usr/include/sys/ide.h** file.

When an **errno** value of **EIO** is received, the caller should evaluate the returned status in the `status_validity` field. The `status_validity` field will have the **ATA_ERROR_STATUS** bit set to indicate that the `ata_status` field is valid. The `status_validity` field will also have the **ATA_ERROR_VALID** bit set to indicate that the `ata_errval` field contains a valid error code.

After one or more attempts, the **IDEIOTUR** operation should return a successful completion, indicating that the device was successfully started. If, after several seconds, the **IDEIOTUR** operation still returns an `ata_status` field set to a check condition status, the device should be skipped.

Note: The IDE adapter device driver performs normal error-recovery procedures during execution of this command.

Return Values

When completed successfully, this operation returns a value of 0. For the **IDEIOTUR** operation, this means the target device has been successfully started and is ready for data access. If unsuccessful, this operation returns a value of -1 and the **errno** global variable is set to 1 of the following values:

Value	Description
EFAULT	Indicates that a bad copy between kernel and user space occurred.
EINVAL	Indicates the **IDEIOSTART** operation was not issued prior to this command.
EIO	Indicates the adapter device driver was unable to complete the command due to an unrecoverable **I/O** error. If **EIO** is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. Following an unrecovered **I/O** error, the adapter error status information is logged in the system error log.
ENOCONNECT	Indicates a bus fault has occurred. In general, the IDE adapter device driver cannot determine which device caused the IDE bus fault, so this error is not logged.
ENODEV	Indicates no IDE device responded to the requested IDE device ID. This condition is not necessarily an error and is not logged.
ENOMEM	Indicates insufficient memory is available to complete the command.
ETIMEDOUT	Indicates the device did not respond with a status before the internal command time-out value expired. If this return value is received, the caller should retry this command at least once, as the first command may have cleared an error condition with the device. This error is logged in the system error log.

Files

/dev/ide0, /dev/ide1,..., /dev/ide*n*	Provide an interface to allow IDE device drivers to access IDE devices or adapters.

Related Information

idedisk IDE device driver or idecdrom IDE device driver.

Chapter 7. SSA Subsystem

SSA Subsystem Overview

Device Drivers

Two types of device driver provide support for all SSA subsystems:

- The SSA adapter device driver, which deals with the SSA adapter.
- The SSA head device drivers, which deal with devices that are attached to the SSA adapter. The SSA disk device driver is an example of an SSA head device driver.

For subsytems that use Micro Channel SSA Multi-Initiator/RAID EL Adapters or PCI SSA Multi-Initiator/RAID EL Adapters, the Target-Mode SSA (TMSSA) device driver is also available. This device driver provides support for communications from using system to using system. For information about SSA Target Mode and the TMSSA device driver, see SSA Target Mode.

Note: Micro Channel machines will only run AIX 5.1 or earlier.

Responsibilities of the SSA Adapter Device Driver

The SSA adapter device driver provides a consistent interface to all SSA head device drivers, of which the SSA disk device driver is an example.

The SSA adapter device driver sends commands for SSA devices to the adapter that is related to those devices. When the SSA adapter device driver detects that the commands have completed, it informs the originator of the command.

Responsibilities of the SSA Disk Device Driver

The SSA disk device driver provides support for the SSA disk drives that are connected to an SSA adapter. That support consists of:

- Standard block I/O to SSA logical disks, which are represented as hdisks
- Character mode I/O to SSA logical disks, which are represented as rhdisks
- Error reporting from SSA physical disks, which are represented as pdisks
- Diagnostics and service interface to SSA physical disks that are represented as pdisks
- Re-issue of commands in the event of an adapter reset

Interface between the SSA Adapter Device Driver and Head Device Driver

To communicate with the SSA adapter device driver, the SSA head device driver:

1. Uses the **fp_open** kernel service to open the required instance of the SSA adapter device driver.
2. Calls the **fp_ioctl** kernel service to issue the **SSA_GET_ENTRY_POINT** operation to the opened adapter.
3. Calls the function **SSA_Ipn_Directive** whose address was returned by the **ioctl** operation. These calls to **SSA_Ipn_Directive** are used for all communication with the SSA device.
4. Uses the **fp_close** kernel service to close the adapter.

Note: When **fp_close** is called, **SSA_Ipn_Directive** cannot be called.

Trace Formatting

The SSA adapter device driver and the SSA disk device driver can both make entries in the kernel trace buffer. The hook ID for the SSA adapter device driver is 45A. The hook ID for the SSA disk device driver is

45B. For information on how to use the kernel trace feature, refer to the **trace** command for the kernel debug program. With the PCI SSA Multi-Initiator/RAID EL Adapter and Micro Channel Enhanced SSA Multi-Initiator/RAID EL Adapter, the Target-Mode SSA device driver can make entries in the kernel trace buffer; its hook ID is xxx.

Related Information
The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver.

Trace Command for the Kernel Debug Program.

SSA Adapter Device Driver

Purpose
Supports the SSA adapter.

Syntax
#include </usr/include/sys/ssa.h>
#include </usr/include/sys/devinfo.h>

Description
The **/dev/ssa***n* special files provide an interface that allows client application programs to access SSA adapters and the SSA devices that are connected to those adapters. Multiple-head device drivers and application programs can all access a particular SSA adapter and its connected devices at the same time.

Configuring Devices
All the SSA adapters that are connected to the using system are normally configured automatically during the system boot sequence.

SSA Micro Channel Adapter ODM Attributes

Note: A Micro Channel Machine can only run AIX 5.1 or earlier.

The SSA Micro Channel adapter has a number of object data manager (ODM) attributes that you can display by using the **lsattr** command:

ucode	Holds the file name of the microcode package file that supplies the adapter microcode that is present in an SSA adapter.
bus_intr_level	Holds the value of the bus interrupt level that the SSA adapter device driver for this adapter will use.
dma_lvl	Holds the value of the DMA arbitration level that the SSA adapter device driver for this adapter will use.
bus_io_addr	Holds the value of the bus I/O base address of the adapter registers that the SSA adapter device driver for this adapter will use.
dma_bus_mem	Holds the value of the bus I/O base address of the adapter's DMA address that the SSA adapter device driver for this adapter will use.
dbmw	Holds the size of the DMA area that the SSA adapter device driver for this adapter will use. You can use the **chdev** command to change the value of this attribute. The default value provides a DMA area that is large enough to allow the adapter to perform efficiently, yet allows other adapters to be configured. The default value is practical for normal use. If, however, a particular SSA device that is attached to the using system needs large quantities of outstanding I/O to get best performance, a larger DMA area might improve the performance of the adapter.
bus_mem_start	Holds the value of the bus-memory start address that the SSA adapter device driver for this adapter will use.

intr_priority Holds the value of the interrupt priority that the SSA adapter device driver for this adapter will use.

daemon Specifies whether to start the SSA adapter daemon. If the attribute is set to TRUE, the daemon is started when the adapter is configured.

The daemon holds the adapter device driver open although the operating system might not be using that adapter device driver at the time. This action allows the adapter device driver to reset the adapter card if the software that is running on it finds an unrecoverable problem. It also allows the adapter device driver to log errors against the adapter.

The ability of the device driver to log errors against the adapter is especially useful if the adapter is in an SSA loop that is used by another adapter, because failure of this adapter can affect the availability of the SSA loop to the other adapter.

You can use the **chdev** command to change the value of this attribute.

host_address This attribute may be used to specify the TCPIP address used by the SSA network agent on remote hosts to contact this host. If set, the value is passed to remote hosts via the SSA network. If this attribute is not set then the value returned by the ″hostname″ command is passed to remote hosts.

This may be useful on systems which have more than one tcpip address and where the specific TPCIP address used by the SSA network agent is important.

This attribute is only functional for the PCI SSA Multi-Initiator/RAID EL Adapter and the Micro Channel SSA Multi-Initiator/RAID EL Adapter.

PCI SSA Adapter ODM Attributes

The PCI SSA adapter has a number of object data manager (ODM) attributes that you can display by using the **lsattr** command:

ucode Holds the file name of the microcode package file that supplies the adapter microcode that is present in an SSA adapter.

bus_intr_level Holds the value of the bus interrupt level that the SSA adapter device driver for this adapter will use.

bus_io_addr Holds the value of the bus I/O base address of the adapter registers that the SSA adapter device driver for this adapter will use.

bus_mem_start Holds the value of the bus-memory start address that the SSA adapter device driver for this adapter will use.

bus_mem_start2 Holds the value of the bus-memory start address that the SSA adapter device driver for this adapter will use.

intr_priority Holds the value of the interrupt priority that the SSA adapter device driver for this adapter will use.

daemon Specifies whether to start the SSA adapter daemon. If the attribute is set to TRUE, the daemon is started when the adapter is configured.

The daemon holds the adapter device driver open although the operating system might not be using that adapter device driver at the time. This action allows the adapter device driver to reset the adapter card if the software that is running on it finds an unrecoverable problem. It also allows the adapter device driver to log errors against the adapter.

The ability of the device driver to log errors against the adapter is especially useful if the adapter is in an SSA loop that is used by another adapter, because failure of this adapter can affect the availability of the SSA loop to the other adapter.

You can use the **chdev** command to change the value of this attribute.

Device-Dependent Subroutines

The SSA adapter device driver provides support only for the **open**, **close**, and **ioctl** subroutines. It does not provide support for the **read** and **write** subroutines.

open and close Subroutines

The **open** and **openx** subroutines must be called by any application program that wants to send **ioctl** calls to the device driver.

You can use the **open** or the **openx** subroutine call to open the SSA adapter device driver. If you use the **openx** subroutine call, set the *ext* parameter to 0, because the call does not use it.

Summary of SSA Error Conditions

If an **open** or **ioctl** subroutine that has been issued to an SSA adapter fails, the subroutine returns -1, and the global variable **errno** is set to a value from the file **/usr/include/sys/errno.h**.

Possible **errno** values for the SSA adapter device driver are:

EINVAL	An unknown ioctl was attempted or the parameters supplied were not valid.
EIO	An I/O error occurred.
ENOMEM	The command could not be completed because not enough real memory or paging space was available.
ENXIO	The requested device does not exist.

Managing Dumps

The SSA adapter device driver is a target for the system dump facility.

The **DUMPQUERY** option returns a minimum transfer size of 0 bytes and a maximum transfer size that is appropriate for the SSA adapter.

To be processed, calls to the SSA adapter device driver **DUMPWRITE** option should use the *arg* parameter as a pointer to the **SSA_Ioreq_t** structure, which is defined in **/usr/include/sys/ssa.h**. Using this interface, commands for which the adapter provides support can be run on a previously started (opened) target device. The SSA adapter device driver ignores the *uiop* parameter.

Note: Only the `SsaMCB.MCB_Result` field of the **SSA_Ioreq_t** structure is set at completion of the **DUMPWRITE**. During the dump, no support is provided for error logging.

If the **dddump** entry point completes successfully, it returns a 0. If the entry point does not complete successfully, it returns one of the following:

EINVAL	A request that is not valid was sent to the adapter device driver; for example, a request for the **DUMPSTART** option was sent before a **DUMPINIT** option had been run successfully
EIO	The adapter device driver was unable to complete the command because the required resources were not available, or because an I/O error had occurred.
ETIMEDOUT	The adapter did not respond with status before the passed command time-out value expired.

Files

/dev/ssa0, /dev/ssa1,..., /dev/ssa*n*	Provide an interface to allow SSA head device drivers to access SSA devices or adapters.

Related Information

The **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

The **chdev** command. The **lsattr** command.

SSA Adapter Device Driver Direct Call Entry Point

Purpose

This direct call entry point allows another kernel extension to send transactions to the SSA Adapter Device Driver. This is not valid for a user process. On completion the caller will be notified by an off level interrupt. See **SSA_GET_ENTRY_POINT** SSA Adapter ioctl operation.

Description

The entry point address is the address returned in *EntryPoint* by the **SSA_GET_ENTRY_POINT** ioctl operation. The function takes a single parameter of type **SSA_Ioreq_t** which is defined in **/usr/include/sys/ssa.h.**

The fields of the **SSA_Ioreq_t** structure are used as follows:

Field	Description
`SsaDPB`	An array of size SSA_DPB_SIZE which is used by the SSA Adapter Device Driver and should be initialized to all NULLs.
`SsaNotify`	The address of the function in the SSA head device driver which the SSA Adapter Device Driver calls when the directive has completed.
`u0`	This is the transaction to be executed. Valid transactions are described in the *Technical Reference* for the adapter.

Return Values

This function does not return errors. The success or otherwise of the directive can be established by examining the directive status byte and transaction result fields which are set up in the SSA MCB. For details see the *Technical Reference* for tha adapter.

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

IOCINFO (Device Information) SSA Adapter Device Driver ioctl Operation

Purpose

Returns a structure defined in the **/usr/include/sys/devinfo.h** file.

Description

The **IOCINFO ioctl** operation returns a structure that is defined in the **/usr/include/sys/devinfo.h** header file. The caller supplies the address to an area that is of the type `struct devinfo`. This area is in the *arg* parameter to the **IOCINFO** operation. The `device-type` field for this component is **DD_BUS**; the `subtype` is **DS_SDA**.

The **IOCINFO** operation is defined for all device drivers that use the **ioctl** subroutine, as follows:

The operation returns a **devinfo** structure. The caller supplies the address of this structure in the argument to the **IOCINFO** operation. The device type in this structure is **DD_BUS**, and the subtype is **DS_SDA**. The `flags` field is set to **DF_FIXED**.

Files

/dev/ssa0, /dev/ssa1,..., /dev/ssa*n*

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA disk device driver, SSA Subsystem Overview.

SSA_GET_ENTRY_POINT SSA Adapter Device Driver ioctl Operation

Purpose

The **SSA_GET_ENTRY_POINT** operation allows another kernel extension, typically a SSA head device driver, to determine the direct call entry point for the SSA adapter device driver. This operation is the entry point through which the head device driver communicates with the adapter device driver. The address that is supplied is valid only while the calling kernel extension holds an open file descriptor for the SSA adapter device driver. This operation is not valid for a user process.

Description

The *arg* parameter specifies the address of a **SSA_GetEntryPointParms_t** structure in kernel address space. The **SSA_GetEntryPointParms_t** structure is defined in the **/usr/include/sys/ssa.h** file.

On completion of the operation, the fields in the **SSA_GetEntryPointParms_t** structure are modified as follows:

Field	Description
`EntryPoint`	Address of the direct call entry point for the SSA adapter device driver, which is used to submit operations from a head device driver.
`InterruptPriority`	The off level interrupt priority at which the calling kernel extension is called back for completion of commands that are started by calling the direct call entry point.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to the following value:

Value	Description
EINVAL	Indicates that the caller was not in kernel mode.

Files

/dev/ssa0, /dev/ssa1,..., /dev/ssa*n*

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

SSA_TRANSACTION SSA Adapter Device Driver ioctl Operation

Purpose

Sends an SSA transaction to an SSA adapter.

Description

The **SSA_TRANSACTION** operation allows the caller to issue an IPN (Independent Packet Network) transaction to a selected SSA adapter. IPN is the language that is used to communicate with the SSA adapter. The caller must be root, or have an effective user ID of root, to issue this operation.

IPN is described in the *Technical Reference* for the adapter.

The *arg* parameter for the **SSA_TRANSACTION** operation specifies the address of a **SSA_TransactionParms_t** structure. This structure is defined in the **/usr/include/sys/ssa.h** file.

The **SSA_TRANSACTION** operation uses the following fields of the **SSA_TransactionParms_t** structure:

Field	Description
DestinationNode	Contains the target node for the transaction.
DestinationService	Contains the target service on that node.
MajorNumber	Major number of the transaction.
MinorNumber	Minor number of the transaction.
DirectiveStatusByte	Contains the directive status byte for the transaction. This contains a value that is defined in the **/usr/include/ipn/ipndef.h** file. A non-zero value indicates an error.
TransactionResult	Contains the IPN result word that is returned by IPN for the transaction. This contains values that are defined in the **/usr/include/ipn/ipntra.h** file. A non-zero value indicates an error.
ParameterDDR	Set by the caller to indicate the buffer for parameter data.
TransmitDDR	Set by the caller to indicate the buffer for transmit data.
ReceiveDDR	Set by the caller to indicate the buffer for received data.
StatusDDR	Set by the caller to indicate the buffer for status data.
TimeOutPeriod	Number of seconds after which the transaction is considered to have failed. A value of 0 indicates no time limit. **Note:** If an operation takes longer to complete than the specified timeout, the adapter is reset to purge the command.

Attention: This is a very low-level interface. It is for use only by configuration methods and diagnostics software. Use of this interface might result in system hangs, system crashes, system corruption, or undetected data loss.

Return Values

When completed successfully, this operation returns a value of 0. Otherwise, a value of -1 is returned, and the **errno** global variable is set to one of the following values:

Value	Description
EIO	Indicates an unrecoverable I/O error.
ENXIO	Indicates an unknown device.
EINVAL	Indicates an unknown command. Indicates a bad buffer type.
EACCESS	Indicates user does not have root privilege.
ENOMEM	Indicates not enough memory.
ENOSPC	Indicates not enough file blocks.
EFAULT	Indicates bad user address.

Files

/dev/ssa0, /dev/ssa1,..., /dev/ssa*n*

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

ssadisk SSA Disk Device Driver

Purpose

Provides support for Serial Storage Architecture (SSA) disk drives.

Syntax

```
#include <sys/devinfo.h>
#include <sys/ssa.h>
#include <sys/ssadisk.h>
```

Configuration Issues

SSA Logical disks, SSA Physical disks, and SSA RAID Arrays

Serial Storage Architecture (SSA) disk drives are represented as SSA logical disks (**hdisk0**, **hdisk1**.....**hdisk***N*) and SSA physical disks (**pdisk0**,**pdisk1**.....**pdisk***N*). SSA RAID arrays are represented as SSA logical disks (**hdisk0**, **hdisk1**.....**hdisk***N*). SSA logical disks represent the logical properties of the disk drive or array, and can have volume groups and file systems mounted on them. SSA physical disks represent the physical properties of the disk drive.

By default:

- One pdisk is always configured for each physical disk drive.
- One hdisk is configured either for each disk drive that is connected to the using system, or for each array.

By default, all disk drives are configured as system disk drives. The array management software deletes hdisks to create arrays.

SSA physical disks have the following properties:

- configured as **pdisk0**, **pdisk1**.....**pdisk***N*
- Have errors logged against them in the system error log.
- Support a character special file (**/dev/pdisk0**, **/dev/pdisk1**..../**dev/pdisk***N*)
- Support the **ioctl** subroutine for servicing and diagnostics functions.
- Did not accept **read** or **write** subroutine calls for the character special file.

SSA logical disks have the following properties:

- configured as **hdisk0**, **hdisk1**.....**hdisk***N*
- Support a character special file (**/dev/rhdisk0**, **/dev/rhdisk1**..../**dev/rhdisk**N)
- Support a block special file (**/dev/hdisk0**, **/dev/hdisk1**..../**dev/hdisk***N*)
- Support the **ioctl** subroutine call for non service and diagnostics functions only.
- Accept the **read** and **write** subroutine call to the special files.
- Can be members of volume groups and have filesystems mounted upon them.

Multiple Adapters

Some SSA subsystems allow a disk drive to be controlled by up to two adapters in a particular using system. The disk drive has, therefore, two paths to each using system, and the SSA subsystem can continue to function if an adapter fails. If an adapter fails or the disk drive becomes inaccessible from the original adapter, the SSA disk device driver switches to the alternative adapter without returning an error to any working application.

Once a disk drive has been successfully opened, takeover by the alternative adapter does not occur simply because a drive becomes reserved or fenced out. However, during an open of a ssa logical disk,

the device driver does attempt to access the disk drive through the alternative adapter if the path through the original adapter experiences reservation conflict or fenced-out status.

Takeover does not occur because of a medium error on the disk drive.

Takeover occurs only after extensive error-recovery activity within the adapter and several retries by the device driver. Intermittent errors that last for only approximately one second usually do not cause adapter takeover.

Once takeover has successfully occurred and the device driver has accessed the disk drive through the alternative adapter, the original adapter becomes the standby adapter. Takeover can, therefore, occur repeatedly from one adapter to another so long as one takeover event is completed before the next one starts. Completion of a takeover event is considered to have occurred when the device driver successfully accesses the disk drive through the alternative adapter.

Once takeover has occurred, the device driver continues to use the alternative adapter to access the disk drive until either the system is rebooted, or takeover occurs back to the original adapter.

Each time the SSA disks are configured, the SSA disk device driver is informed which path or paths are available to each disk drive, and which adapter is to be used as the primary path. By default, primary paths to disk drives are shared equally among the adapters to balance the load. This static load balancing is performed once, when the devices are configured for the first time. You can use the **chdev** command to modify the primary path.

Because of the dynamic nature of the relationship between SSA adapters and disk drives, SSA pdisks and hdisks are not children of an adapter but of an SSA router. This router is called **ssar**. It does not represent any actual hardware, but exists only to be the parent device for the SSA logical disks and SSA physical disks.

Note: When the SSA disk device driver switches from using one adapter to using the other adapter to communicate with a disk, it issues a command that breaks any SSA-SCSI reserve condition that might exist on that disk. The reservation break is only performed if this host had successfully reserved the disk drive through the original adapter. This check is to prevent adapter takeover from breaking reservations held by other using systems. If multiple using systems are connected to the SSA disks, SSA-SCSI reserve should not, therefore, be used as the only method for controlling access to the SSA disks. Fencing is provided as an alternative method for controlling access to disks that are connected to multiple using systems.

PCI SSA Multi-Initiator/RAID EL Adapters and Micro Channel SSA Multi-Initiator/RAID EL Adapters are capable of reserving to a node number rather than reserving to an adapter. It is highly recommended that you make use of this ability by setting the SSA router node_number attribute if multiple adapters are to be configured as described here.

Configuring SSA disk drive devices.

SSA disk drives are represented as SSA Logical disks (**hdisk0**, **hdisk1**.....**hdisk**N) and SSA physical disks (**pdisk0**,**pdisk1**.....**pdisk***n*). The properties of each are described in the **SSA Subsystem Overview.**

Normally, all the disk drives connected to the system will be configured automatically by the system boot process and the user will need to take no action to configure them.

Since some SSA devices may be connected to the SSA network while the system is running without taking the system off line it may be necessary to configure SSA disks after the boot process has completed. In this case the devices should be configured by running the configuration manager with the **cfgmgr** command.

An exception is to configure a specific device with a specific name. This may be achieved using the **mkdev** command.

Using mkdev to Configure a Physical Disk: To use **mkdev** to configure a SSA physical disk it will be necessary to specify the following information:

Parent	ssar
Class	pdisk
Subclass	ssar
Type	You can list the types by typing: `lsdev -P -c pdisk -s ssar`
ConnectionLocation	15-character unique identity of the disk drive. You can determine the unique identifier in three ways: • If the disk drive has already been defined the unique identity may be determined using the **lsdev** command as follows: 1. Enter `lsdev -Ccpdisk -r connwhere`. 2. Select the 15-character unique identifier for which characters 5 to 12 match those on the front of the disk drive. • Otherwise the 15-character unique identifier can be constructed from the 12-character SSA UID on the label on the side of the disk drive suffixed by the 3 characters ″00D″. • Run the **ssacand** command, and specify the adapter to which the physical disk is connected. For example: `ssacand -a ssa0 -P`

Using mkdev to Configure a Logical Disk: In order to use **mkdev** to configure a SSA logical disk it will be necessary to specify the following information:

Parent	ssar
Class	disk
Subclass	ssar
Type	hdisk

ConnectionLocation 15-character unique identity of the disk drive.

If the logical disk is a system disk, you can determine the unique identifier in three ways:

- If the disk drive has already been defined the unique identity may be determined using the **lsdev** command as follows:
  1. Enter `lsdev -Ccdisk -r connwhere` and press Enter.
  2. Select the 15-character unique identifier for which characters 5 to 12 match the serial number that is on the front of the disk drive.
- Construct 15-character unique identifier can be constructed from the 12-character SSA UID on the label on the side of the disk drive suffixed by the 3 characters ″00D″.
- Run the **ssacand** command, and specify the adapter to which the logical disk is connected. For example:

  ```
  ssacand -a ssa0 -L
  ```

If the logical disk is an array, you can determine the unique identifier in two ways:

- If the logical disk has already been defined, you can use the **lsdev** command to determine the unique identifier, as follows:
  1. Type `lsdev -Ccdisk -r connwhere` and press Enter.
  2. Select the 15-character unique identifier that was given by the RAID configuration program when the array was created.
- Run the **ssacand** command, and specify the adapter to which the logical disk is connected. For example:

  ```
  ssacand -a ssa0 -L
  ```

Device Attributes

SSA logical disks and SSA physical disks and the ssar router, have several attributes.You can use the **lsattr** command to display these attributes.

Attributes of the SSA Router, ssar.

node_number This must be set on systems which are using SSA Fencing or the SSA Disk Concurrent Mode of Operation Interface.

Both of these features of the SSA disk device driver are used only in configurations which have more than one host system connected to the same SSA disk drives. In configurations where only one host system is connected to the SSA disk drives this attribute has no effect.

For configurations using SSA Fencing or the SSA Disk Concurrent Mode of Operation Interface this attribute should be set to a different value on each host in the configuration.
Note: After this attribute has been modified it is necessary to reboot the system for it to take effect.

Attributes which are common to SSA logical and SSA physical disks.

adapter_a Specifies the name of one adapter connected to the device or **none** if no adapter is currently connected as adapter_a.

adapter_b Specifies the name of one adapter connected to the device or **none** if no adapter is currently connected as adapter_b.

primary_adapter Specifies whether adapter_a or adapter_b is to be the primary adapter for this device.

This attribute may be modified using the **chdev** command to one of the values adapter_a, adapter_b or **assign**. If the value is set to **assign**, static load balancing will be performed when this device is made available and the system will set the value to either adapter_a or adapter_b.

connwhere_shad Holds a copy of the value of the connwhere parameter for this disk drive. SSA disks drives cannot be identified by the location field given by **lsdev**. This is because they are connected in a loop and do not have hardware-selectable addresses like SCSI devices. The only means of identification of SSA devices is their serial number and this is written in the connwhere field of the CuDv entry for the device. Providing this **connwhere_shad** attribute, which shadows the *connwhere* value, means the user can display the *connwhere* value for an SSA device for a pdisk or hdisk.

location Describes, in text, the descriptions of the disk drives and their locations (for example, drawer number 1, slot number 1). The information for this attribute is entered by the user.

Attributes for SSA Logical Disks Only

pvid Holds the ODM copy of the PVID for this disk drive for an hdisk.

queue_depth Specifies the maximum number of commands that the SSA disk device driver dispatches for a single disk drive for an hdisk. You can use the **chdev** command to modify this attribute. The default value is correct for normal operating conditions

reserve_lock Specifies whether the SSA disk device driver locks the device with a reservation when it is opened for an hdisk.

size_in_mb Specifies the size of the logical disk in megabytes.

max_coalesce This is the maximum number of bytes which the SSA disk device driver attempts to transfer to or from an SSA logical disk in a single operation. The default value is appropriate for most environments. For applications that perform very long sequential write operations, there are performance benefits in writing data in blocks of 64KB times the number of disks in the array minus one (these are known as *full-stride writes* times the number of disks in the array minus one, or to some multiple of this number.

write_queue_mod Alters the way in which write commands are queued to SSA logical disks. The default value is 0 for all SSA logical disks that do not use the fast-write cache; with this setting the SSA disk device driver maintains a single seek-ordered queue of queue_depth operations on the disk. Reads and writes are queued together in this mode.

If write_queue_mod is set to a non-zero value, the SSA disk device driver maintains two separate seek-ordered queues, one for reads and one for writes. In this mode, the device driver issues up to queue_depth read commands and up to write_queue_mod write commands to the logical disk.

This facility is provided because in some environments it may be beneficial to hold back write commands in the device driver so that they may be coalesced into larger operations which may be handled as full-stride writes by the RAID software within the adapter.

This facility is unlikely to be useful unless a large percentage of the workload to a RAID-5 device is composed of sequential write operations.

Device-Dependent Subroutines

The **open**, **read**, **write**, and **close** subroutines start typical physical volume operations.

open, read, write and close Subroutines

The **open** subroutine is intended primarily for use by the diagnostic commands and utilities. Appropriate authority is required for execution. If an attempt is made to run the **open** subroutine without the proper authority, the subroutine returns a value of -1 and sets the **errno** global variable to a value of **EPERM**.

The *ext* parameter passed to the **openx** subroutine selects the operation to be used for the target device. The **/usr/include/sys/ssadisk.h** file defines possible values for the *ext* parameter.

The *ext* parameter can contain any combination of the following flag values logically ORed together:

SSADISK_PRIMARY
Opens the device using the primary adapter as the path to the device. As a result of hardware errors the device driver may automatically switch to the secondary path if one exists. This can be prevented by additionally specifying the **SSADISK_NOSWITCH** flag.

This flag is supported for both SSA logical disks and SSA physical disk drives.This flag cannot be specified together with **SSADISK_SECONDARY**.

SSADISK_SECONDARY
Opens the device using the secondary adapter as the path to the device. As a result of hardware errors the device driver may automatically switch to the primary path if one exists. This can be prevented by additionally specifying the **SSADISK_NOSWITCH** flag.

This flag is supported for both SSA logical disks and SSA physical disk drives.This flag cannot be specified together with **SSADISK_PRIMARY**.

SSADISK_NOSWITCH
If more than one adapter provides a path to the device, the device driver normally switches from one adapter to the other as part of its error recovery. This flag prevents this from happening.

This flag is supported for both SSA logical disks and SSA physical disk drives.

SSADISK_FORCED_OPEN
Forces the open regardless of whether another initiator has the device reserved. If another initiator has the device reserved, the reservation is broken. In other respects, the **open** operation runs normally.

This flag is supported only for SSA logical disks. This flag cannot be specified together with **SSADISK_FENCEMODE**.

SSADISK_RETAIN_RESERVATION
Retains the reservation of the device after a **close** operation by not issuing the release. This flag prevents other initiators from using the device unless they break the host machine's reservation.
Note: This does not cause the device to be explicitly reserved during the close if it was not reserved while it was open.

This flag is supported only for SSA logical disk drives. This flag cannot be specified together with **SSADISK_FENCEMODE**.

SSADISK_NO_RESERVE
Prevents the reservation of a device during an **openx** subroutine call to that device. This operation is provided so a device can be controlled by two processors that synchronize their activity by their own software means.

This flag overrides the setting of the attribute reserve_lock if the value of the attribute is yes. This flag is supported only for SSA logical disk drives. This flag cannot be specified together with **SSADISK_FENCEMODE**.

SSADISK_SERVICEMODE Opens an SSA physical disk in service mode. This wraps the SSA links either size of the indicated physical disk allowing it to be removed from the loop for service without causing errors on the loops.

This flag is supported only for SSA physical disk drives. This flag cannot be specified together with **SSADISK_SCSIMODE**.

SSADISK_SCSIMODE Opens an SSA physical disk in SCSI passthrough mode. This allows **SSADISK_IOCTL_SCSI** ioctls to be issued to the physical disk.

This flag is supported only for SSA physical disk drives. This flag cannot be specified together with **SSADISK_SERVICEMODE**.

SSADISK_NORETRY Opens a device in no-retry mode.

When a device is opened in this mode, commands are not retried if an error occurs.

SSADISK_FENCEMODE Opens an SSA logical disk drive in fence mode. The open succeeds even if the host is fenced out from access to the disk drive. Only ioctls can be issued to the device while it is open in this mode. Any attempt to read from or write to a device opened in this mode will be rejected with an error.

This flag is supported only for SSA logical disk drives. This flag cannot be specified together with **SSADISK_NO_RESERVE**, **SSADISK_FORCED_OPEN** or **SSADISK_RETAIN_RESERVATION**.

″SSA Options to the openx Subroutine″ in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts* gives more specific information on the open operations.

readx and writex Subroutines

The **readx** and **writex** subroutines provide additional parameters affecting the raw data transfer. These subroutines pass the *ext* parameter, which specifies request options. The options are constructed by logically ORing zero or more of the following values:

HWRELOC Indicates a request for hardware relocation (safe relocation only).
UNSAFEREL Indicates a request for unsafe hardware relocation.
WRITEV Indicates a request for write verification.

Error Conditions

Possible **errno** values for **ioctl**, **open**, **read**, and **write** subroutines when the SSA device driver is used, include:

EBUSY Indicates one of the following circumstances:
- An attempt was made to open an SSA physical device which is already opened by another process.
- The target device is reserved by another initiator.

EFAULT Indicates an illegal user address.

EINVAL Indicates one of the following circumstances:

- The **read** or **write** subroutine supplied an *nbyte* parameter that is not an even multiple of the block size.
- The data buffer length exceeded the maximum defined in the **devinfo** structure for an **ioctl** subroutine operation.
- The **openext** subroutine supplied an unsupported combination of extension flags.
- An unsupported **ioctl** subroutine operation was attempted.
- An attempt was made to configure a device that is still open.
- An illegal configuration command has been given.
- The data buffer length exceeded the maximum defined for a **strategy** operation.

EIO Indicates one of the following circumstances:

- The target device cannot be located or is not responding.
- The target device has indicated an unrecovered hardware error.

ESOFT Indicates that the target device has reported a recoverable media error.

EMEDIA Indicates that the target device has encountered an unrecovered media error.

ENODEV Indicates one of the following circumstances:

- An attempt was made to access an undefined device.
- An attempt was made to close an undefined device.

ENOTREADY Indicates that an attempt was made to open a SSA physical device in service mode whilst a SSA logical device which uses it was in use.

ENXIO Indicates one of the following circumstances:

- The **ioctl** subroutine supplied an invalid parameter.
- The **openext** subroutine supplied extension flags which selected a non-existent or non-functional adapter path.
- A **read** or **write** operation was attempted beyond the end of the fixed disk drive.

EPERM Indicates the attempted subroutine requires appropriate authority.

ENOCONNECT Indicates that the host has been fenced out from access to this device.

ENOMEM Indicates that the system has insufficient real memory or insufficient paging space to complete the operation.

ENOLCK Indicates that an attempt was made to open a device in service mode which is in an SSA network which is not a loop.

Special Files

The **ssadisk** device driver uses raw and block special files in performing its functions.

> **Attention:** Data corruption, loss of data, or loss of system integrity (system crash) will occur if devices supporting paging, logical volumes, or mounted file systems are accessed using block special files. Block special files are provided for logical volumes and disk devices and are solely for system use in managing file systems, paging devices, and logical volumes. These files should not be used for other purposes.

The special files used by the **ssadisk** device driver include the following (listed by type of device):

- SSA logical disk drives:

/dev/hdisk0, /dev/hdisk1,..., /dev/hdisk*n* Provide an interface to allow SSA device drivers block I/O access to logical SSA disk drives.

/dev/rhdisk0, **/dev/rhdisk1**,..., **/dev/rhdisk***n* Provide an interface to allow SSA device drivers character access (raw I/O access and control functions) to logical SSA disk drives.

- SSA physical disk drives:

/dev/pdisk0, **/dev/pdisk1**, **...**, **/dev/pdisk***n*	Provide an interface to allow SSA device drivers character access (control functions only) to physical SSA disks drives.

Note: The prefix **r** on a special file name indicates the drive is accessed as a raw device rather than a block device. Performing raw I/O with an SSA logical disk requires that all data transfers be in multiples of the device block size. Also, all **lseek** subroutines that are made to the raw device driver must result in a file pointer value that is a multiple of the device block size.

Related Information

Special Files Overview in *AIX 5L Version 5.2 Files Reference*.

Understanding the Execution of Initiator I/O Requests in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

SCSI Error Recovery in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Understanding the sc_buf Structure in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

The **rmdev** command. The **mkdev** command. The **cfgmgr** command. The **chdev** command. The **lsdev** command. The **lsattr** command.

The **close** subroutine, **ioctl or ioctlx** subroutine, **open, openx, or creat** subroutine, **read, readx, readv, or readvx** subroutine,**write, writex, writev, or writevx** subroutine.

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

IOCINFO (Device Information) SSA Disk Device Driver ioctl Operation

Purpose

Returns a structure defined in the **/usr/include/sys/devinfo.h** file.

Description

The **IOCINFO** operation returns a structure defined in the **/usr/include/sys/devinfo.h** header file. The caller supplies the address to an area of type `struct devinfo` in the *arg* parameter to the **IOCINFO** operation. The `device-type` field for this component is **DD_SCDISK**; the `subtype` is **DS_PV**. The information returned includes the block size in bytes and the total number of blocks on the disk drive.

Files

/dev/pdisk0, **/dev/pdisk1**, **...**, **/dev/pdisk***n*	Provide an interface to allow SSA device drivers to access SSA physical disks drives.
/dev/pdisk0, **/dev/pdisk1**,**...**, **/dev/pdisk***n*	Provide an interface to allow SSA device drivers to access SSA logical disks drives.

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

SSADISK_ISALMgr_CMD (ISAL Manager Command) SSA Disk Device Driver ioctl Operation

Purpose

Provides a means to send Independent Network Storage Access Language (ISAL) Manager commands to an SSA physical or logical disk drive. ISAL comprises a set of commands which allow a program to control and access a storage device. The ISAL Command set is described in the *Technical Reference* for the adapter.

Description

The **SSADISK_ISALMgr_CMD** operation allows the caller to issue an ISAL command to a selected logical or physical disk. The caller must be root or have an effective user ID of root to issue this ioctl.

The following ISAL commands (minor function codes), defined in **/usr/include/ipn/ipnsal.h** can be issued:

FN_ISALMgr_Inquiry
FN_ISALMgr_GetPhysicalResourceIDs
FN_ISALMgr_Characteristics
FN_ISALMgr_FlashIndicator
FN_ISALMgr_HardwareInquiry
FN_ISALMgrVPDInquiry
FN_ISALMgr_Statistics

The *arg* parameter for the **SSADISK_ISALMgr_CMD** ioctl is the address of an **ssadisk_ioctl_parms** structure. This structure is defined in the **/usr/include/sys/ssadisk.h** file.

The **SSADISK_ISALMgr_CMD** ioctl uses the following fields of the **ssadisk_ioctl_parms** structure:

Field	Description
`dsb`	Contains the directive status byte returned for the command. This contains a value from **/usr/include/ipn/ipndef.h.** A non zero value indicates an error.
`result`	Contains the IPN result word returned by IPN for the command. This contains values from **/usr/include/ipn/ipntra.h.** A non zero value indicates an error.
`u0.isal.parameter_descriptor`	Set by the caller to indicate the buffer for parameter data.
`u0.isal.transmit_descriptor`	Set by the caller to indicate the buffer for transmit data.
`u0.isal.receive_descriptor`	Set by the caller to indicate the buffer for received data.
`u0.isal.status_descriptor`	Set by the caller to indicate the buffer for status data.
`u0.isal.minor_function`	Set by the caller to one of the ISAL Manager Commands defined in **/usr/include/ipn/ipnsal.h** and listed above.

Note: Structures are provided in **/usr/include/ipn/ipnsal.h** this can be used to format the contents of the parameter buffer for the various commands. In all cases, the resource ID which is located in the first four bytes of the parameter buffer will be overwritten by the device driver with the correct Resource ID for the device.

Return Values

If the command was successfully sent to the adapter card, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable set to one of the following values:

Value	Description
EIO	Indicates an unrecoverable I/O error.

Value	Description
EINVAL	Indicates that the caller has specified an ISAL manager command that is not in the list of supported ISAL manager commands above.
EPERM	Indicates that caller did not have an effective user ID (EUID) of 0.
ENOMEM	Indicates that the device driver was unable to allocate or pin enough memory to complete the operation.

If the return code is 0, the result field of the **ssadisk_ioctl_parms** structure is valid. This indicates whether the adapter was able to process the command successfully.

Files

/dev/pdisk0, **/dev/pdisk1**,..., **/dev/pdisk***n*	Provide an interface to allow SSA device drivers to access physical SSA disks.
/dev/hdisk0, /dev/hdisk1,..., /dev/hdisk*n*	Provide an interface to allow SSA device drivers to access logical SSA disks.

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

SSADISK_ISAL_CMD (ISAL Command) SSA Disk Device Driver ioctl Operation

Purpose

Provides a means to send Independent Network Storage Access Language (ISAL) commands to an SSA physical or logical disk drive. ISAL comprises a set of commands which allow a program to control and access a storage device. The ISAL Command set is described in the *Technical Reference* for the adapter.

Description

The **SSADISK_ISAL_CMD** operation allows the caller to issue an ISAL command to a selected logical or physical disk drive. The caller must be root or have an effective user ID of root to issue this ioctl.

The following ISAL commands (minor function codes), defined in **/usr/include/ipn/ipnsal.h** may be issued:

FN_ISAL_Read
FN_ISALWrite
FN_ISAL_Format
FN_ISAL_Progress
FN_ISAL_Lock
FN_ISAL_Unlock
FN_ISAL_Test
FN_ISAL_SCSI
FN_ISAL_Download
FN_ISAL_Fence

Note:

1. Some of these commands are not valid for one or other of SSA hdisks or SSA pdisks. This is not checked by the device driver but by the adapter card. If the caller attempts to send a command to a device for which it is not valid, the result returned by the adapter will be non-zero. The exception to this is that the device driver will reject with **EINVAL** any attempt to send a **FN_ISAL_Fence** command to a SSA physical disk.

2. The **FN_ISAL_SCSI** command is rejected by the adapter with a non-zero result if it is sent to a device that has not been opened with the **SSADISK_SCSIMODE** extension parameter.

The *arg* parameter for the **SSADISK_ISAL_CMD** ioctl is the address of an **ssadisk_ioctl_parms** structure. This structure is defined in the **/usr/include/sys/ssadisk.h** file.

The **SSADISK_ISAL_CMD** ioctl uses the following fields of the **ssadisk_ioctl_parms** structure:

Field	Description
`dsb`	Contains the directive status byte returned for the command. This contains a value from **/usr/include/ipn/ipndef.h.** A non zero value indicates an error.
`result`	Contains the Independent Packet Network (IPN) result word returned by IPN for the command. This contains values from **/usr/include/ipn/ipntra.h.** A non-zero value indicates an error.
`u0.isal.parameter_descriptor`	Set by the caller to indicate the buffer for parameter data.
`u0.isal.transmit_descriptor`	Set by the caller to indicate the buffer for transmit data.
`u0.isal.receive_descriptor`	Set by the caller to indicate the buffer for received data.
`u0.isal.status_descriptor`	Set by the caller to indicate the buffer for status data.
`u0.isal.minor_function`	Set by the caller to one of the ISAL commands defined in **/usr/include/ipn/ipnsal.h** and listed above.

Note: Structures are provided in **/usr/include/ipn/ipnsal.h** that can be used to format the contents of the parameter buffer for the various commands. In all cases, the handle which is located in the first four bytes of the parameter buffer will be overwritten by the device driver with the correct handle for the device.

Return Values

If the command was successfully sent to the adapter card this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable set to one of the following values:

Value	Description
EIO	Indicates an unrecoverable I/O error.
EINVAL	Indicates either that the caller has specified an ISAL command that is not in the list of supported ISAL commands, or that the caller has attempted to send a **FN_ISAL_FENCE** command to an SSA physical disk.
EPERM	Indicates that caller did not have an effective user ID (EUID) of 0.
ENOMEM	Indicates that the device driver was unable to allocate or pin enough memory to complete the operation.

If the return code is 0, the result field of the **ssadisk_ioctl_parms** structure is valid. This indicates whether the adapter was able to process the command successfully.

Files

/dev/pdisk0, **/dev/pdisk1**,..., **/dev/pdisk***n*	Provide an interface to allow SSA device drivers to access SSA physical disk drives.
/dev/hdisk0, **/dev/hdisk1,...**, **/dev/hdisk***n*	Provide an interface to allow SSA device drivers to access SSA logical disk drives.

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

SSADISK_SCSI_CMD (SCSI Command) SSA Disk Device Driver ioctl Operation

Purpose

Provides a means to send SSA-SCSI Serial Storage Architecture-Small Computer Systems Interface (SSA-SCSI) commands to an SSA physical disk drive that has been opened with the **SSADISK_SCSIMODE** extension flag.

Description

The **SSADISK_SCSI_CMD** operation allows the caller to issue a SSA-SCSI command to a selected physical disk. The caller must be root or have an effective user ID of root to issue this ioctl.

The *arg* parameter for the **SSADISK_ISALMgr_CMD** operation is the address of an **ssadisk_ioctl_parms** structure. This structure is defined in the **/usr/include/sys/ssadisk.h** file.

The **SSADISK_SCSI_CMD** operation uses the following fields of the **ssadisk_ioctl_parms** structure:

Field	Description
`dsb`	Contains the directive status byte returned for the command. This contains value from **/usr/include/ipn/ipndef.h.** A non zero value indicates an error.
`result`	Contains the IPN result word returned by IPN for the command. This contains values from **/usr/include/ipn/ipntra.h.** A non zero value indicates an error.
`u0.scsi.data_descriptor`	Set by the caller to describe the buffer for any data transferred by the scsi command. If no data is transferred then the length of the buffer should be set to 0.
`u0.scsi.direction`	Set by the caller to indicate the direction of the transfer. Valid values are: **SSADISK_SCSI_DIRECTION_NONE** No data transfer is involved for the command. **SSADISK_SCSI_DIRECTION_READ** Data is transferred from the subsystem into host memory. **SSADISK_SCSI_DIRECTION_WRITE** Data is transferred from host memory into the subsystem.
`u0.scsi.identifier`	Identifies the SSA-SCSI logical unit number to which the command should be sent. The format of this field is as defined for SSA_SCSI (bit 7=1 identifies the Target routine, bits 6-0 identify the Logical Unit routine).
`u0.scsi.cdb`	Set by the caller to define the SCSI Command Descriptor Block (CDB) for the command.
`u0.scsi.cdb_length`	Set by the caller to indicate the length of the CDB.
`u0.scsi.scsi_status`	Contains the SCSI status returned for the command.

The device driver has no knowledge of the contents of the CDB, simply passing it on to the hardware. The user should consult the relevant hardware documentation to determine what CDBs are valid for a particular SSA physical disk.

Return Values

If the command was successfully sent to the adapter card then this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable set to one of the following values:

Value	Description
EIO	Indicates either an unrecoverable I/O error or that the scsi command was not recognized as valid by the hardware.
EINVAL	The `u0.scsi.cdb_length` field in the **ssadisk_ioctl_parms** structure was set to an invalid length or the `u0.scsi.direction` field in the **ssadisk_ioctl_parms** structure was set to an invalid value.
EPERM	Indicates that caller did not have an effective user ID (EUID) of 0.
ENOMEM	Indicates that the device driver was unable to allocate or pin enough memory to complete the operation.

If the return code is 0, the result field of the **ssadisk_ioctl_parms** structure is valid. This indicates whether the adapter was able to process the command successfully.

Files

/dev/pdisk0, **/dev/pdisk1**,..., **/dev/pdisk***n*	Provide an interface to allow SSA device drivers to access physical SSA disks.
/dev/hdisk0, /dev/hdisk1,..., /dev/hdisk*n*	Provide an interface to allow SSA device drivers to access logical SSA disks.

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

SSADISK_LIST_PDISKS SSA Disk Device Driver ioctl Operation

Purpose

Provides a means to determine which SSA physical disk drives make up a SSA logical disk drive.

Description

The **SSADISK_LIST_PDISKS** operation may be issued by any user to a SSA logical disk (hdisk). It returns a list of the SSA physical disks (pdisks) which make up the specified logical disk drive.

The *arg* parameter for the **SSADISK_LIST_PDISKS** operation is the address of an **ssadisk_ioctl_parms** structure. This structure is defined in the **/usr/include/sys/ssadisk.h** file.

The **SSADISK_LIST_PDISKS** operation uses the following fields of the **ssadisk_ioctl_parms** structure:

Field	Description
`u0.list_pdisks.name_array`	Pointer to array of **ssadisk_name_desc_t** structures in the caller's memory. It is this array which is filled in with the names of the hdisks on return from the ioctl.
`u0.list_pdisks.name_array_elements`	Set by the caller to indicate the number of elements in the array pointed at by the `u0.list_pdisks.name_array` parameter.
`u0.list_pdisks.name_count`	On return from the ioctl, this indicates the number of names in the name array pointed at by `u0.list_pdisks.name_array`.

Field	Description
u0.list_pdisks.resource_count	On return from the ioctl this indicates the number of physical disk drives which make up the logical disk drive. This may be less than u0.list_pdisks.name_count if not enough elements were allocated in the name array in the user's memory to hold all the pdisk names, or one or more of the physical disks which make up the logical disk have not been configured as operating system physical disk drives.

Return Values

If the command was successfully sent to the adapter card, this operation returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable set to one of the following values:

Value	Description
EIO	Indicates an unrecoverable I/O error.
ENOMEM	Indicates that the device driver was unable to allocate or pin enough memory to complete the operation.

Files

/dev/pdisk0, **/dev/pdisk1**,..., **/dev/pdisk***n*	Provide an interface to allow SSA device drivers to access SSA physical disks.
/dev/hdisk0, /dev/hdisk1,..., /dev/hdisk*n*	Provide an interface to allow SSA device drivers to access SSA logical disks.

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

SSA Disk Concurrent Mode of Operation Interface

The SSA subsystem supports the ability to broadcast one-byte message codes from one host to all other hosts connected to the same disk drive. This message-passing capability can be used to synchronize access to the disk drive. The operating system has a concurrent mode interface to use this hardware functionality.

The concurrent mode of operation requires that a top kernel extension runs on all hosts sharing a disk drive. The top kernel extensions communicate with each other via the SSA subsystem using the concurrent mode interface of the SSA disk device driver. This interface allows a top kernel extension to send and receive messages between hosts.

The concurrent mode interface consists of an entry point in both the SSA disk device driver and the top kernel extension. Two ioctls register and unregister the top kernel extension with the SSA disk device driver. The SSA disk device driver's entry point provides the means to send messages as well as lock, unlock, and test disk drive. The top kernel extension entry point processes interrupts, including receiving messages from other hosts.

Note: In order for the concurrent mode interface to work, the **node_number** attribute of the ssar router must be set to a different, non zero, value on each of the hosts sharing a disk drive. After the **node_number** has been assigned, the host must be rebooted for it to take effect.

Device Driver Entry Point

The SSA disk device driver concurrent mode entry point sends commands from the top kernel extension for a specified SSA Disk. The top kernel extension calls this entry point directly. The **DD_CONC_REGISTER** ioctl operation registers entry points.

This entry point function takes one argument, which is a pointer to a **conc_cmd** structure, that is defined in the **/usr/include/sys/ddconc.h** file. The **conc_cmd** structures must be allocated by the top kernel extension. The concurrent mode command operation is specified by the `cmd_op` field in the **conc_cmd** structure and can have the following values. For each operation, the `devno` field of the **conc_cmd** structure specifies the appropriate SSA disk drive. The concurrent mode command operation can have the following values:

Value	Description
DD_CONC_SEND_REFRESH	Broadcasts the one-byte message code specified by the `message` field of the **conc_cmd** structure. The code is sent to all hosts connected to the SSA disk drive.
DD_CONC_LOCK	Locks the specified SSA disk drive for this host only. No other hosts will be able to modify data on the disk drive.
DD_CONC_UNLOCK	Unlocks the SSA disk drive. Other hosts can lock and modify data on the disk drive.
DD_CONC_TEST	Issues a test disk command to verify that the SSA disk drive is still accessible to this host.

The concurrent mode entry point returns a value of **EINVAL** if any of the following are true:

- The top kernel extension did not perform a **DD_CONC_REGISTER** operation.
- The **conc_cmd** pointer is null.
- The `devno` field in the **conc_cmd** structure is invalid.
- The `cmd_op` field in the **conc_cmd** structure is not one of the four valid values previously listed.

If the concurrent mode entry point accepts the **conc_cmd** structure, the entry point returns a value of 0. If the SSA disk device driver does not have resources to issue the command, the driver queues the command until resources are available. The concurrent commands queued in the SSA disk device driver are issued before any read or write operations queued by the driver's strategy entry point.

The completion status of the concurrent mode commands are returned to the top kernel extension's concurrent mode interrupt handler entry point.

Top Kernel Extension Entry Point

The top kernel extension must have a concurrent mode command interrupt handler entry point, which is called directly from the SSA disk device driver's interrupt handler. This function can take four arguments: the **conc_cmd** pointer, and the `cmd_op`, `message_code`, and `devno` fields. The **conc_cmd** pointer points to a **conc_cmd** structure. These arguments must be of the same type specified by the `conc_intr_addr` function pointer field in the **dd_conc_register** structure.

The following valid concurrent mode commands are defined in the **/usr/include/sys/ddcon.h** file. For each, the `devno` field specifies the appropriate SSA disk drive.

Command	Description
DD_CONC_SEND_REFRESH	Indicates the **DD_CONC_SEND_REFRESH** Device Driver entry point completed. The error field in the **conc_cmd** structure contains the return code necessary for the completion of this command. The possible values are defined in the **/usr/include/sys/errno.h** file. The **conc_cmd** pointer argument to the top kernel extension's special interrupt handler entry point is non-null. The `cmd_op`, `message_code`, and `devno` fields are 0.

Command	Description
DD_CONC_LOCK	Indicates the **DD_CONC_SEND_LOCK** device driver entry point completed. The error field in the **conc_cmd** structure contains the return code necessary for the completion of this command. The possible values are defined in the **/usr/include/sys/errno.h** file. The **conc_cmd** pointer argument to the top kernel extension's special interrupt handler entry point is non-null. The `cmd_op`, `message_code`, and `devno` fields are zero.
DD_CONC_UNLOCK	Indicates the **DD_CONC_UNLOCK** device driver entry point completed. The error field in the **conc_cmd** structure contains the return code necessary for the completion of this command. The possible values are defined in the **/usr/include/sys/errno.h** file. The **conc_cmd** pointer argument to the top kernel extension's special interrupt handler entry point is non-null. The `cmd_op`, `message_code`, and `devno` fields are zero.
DD_CONC_TEST	Indicates the **DD_CONC_TEST** device driver entry point completed. The error field in the **conc_cmd** structure contains the return code necessary for the completion of this command. The possible values are defined in the **/usr/include/sys/errno.h** file. The **conc_cmd** pointer argument to the top kernel extension's special interrupt handler entry point is non-null. The `cmd_op`, `message_code`, and `devno` fields are zero.
DD_CONC_RECV_REFRESH	Indicates a message with **message_code** was received for the SSA disk drive specified by the **devno** argument. The **conc_cmd** argument is null for this operation.
DD_CONC_RESET	Indicates the SSA disk drive specified by the **devno** argument was reset, and all pending messages or commands have been flushed. The argument **conc_cmd** is null for this operation.

- The concurrent command interrupt handler routine must have a short path length because it runs on the SSA disk device driver interrupt level. If substantial command processing is needed, then this routine should schedule an off-level interrupt to its own off-level interrupt handler.
- The top kernel extension must have an interrupt priority no higher than the SSA disk device driver's interrupt priority.
- The concurrent command interrupt handler routine might need to disable interrupts at INTCLASS0 if it is expected to use concurrent mode on SSA disk drive and some disks of different types. The other type of disk needs its own device driver to support the concurrent mode.
- A kernel extension that uses the **DD_CONC_REGISTER** ioctl must issue a **DD_CONC_UNREGISTER** ioctl before closing the SSA disk drive.

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

SSA Disk Fencing

SSA disk fencing is a facility which is provided in the SSA subsystem to allow multiple hosts to control access to a common set of disks.

Using the fencing commands provided by the hardware it is possible to exclude individual hosts from accessing a particular disk. The access list for different disks are independent of one another.

Fencing is essentially a function provided by the hardware and manipulated using the hardware commands, but the device driver does have some involvement.

The SSA disk device driver supports fencing by allowing the **FN_ISAL_FENCE** command, which is defined in the *Technical Reference* for the adapter, to be issued to SSA logical disks using the **SSADISK_ISALCMD** ioctl operation.

In order to use fencing, the **node_number** attribute of the **ssar** router must be set to a different value on each host which is participating in fencing. Note that after setting **node_number** the host must be rebooted for the new value to take effect.

By default, the value of **node_number** is 0. This value has particular significance because it is not possible to exclude a host with node number 0 from access to the disk. Thus if a disk is moved from a machine which has been using fencing to a machine which is not involved in fencing, the new machine will be able to communicate with the disk.

If a host attempts to open a disk from which it has been denied access using the **open** subroutine, the return code will be -1 and the global variable **errno** will be set to the value **ENOCONNECT.** Likewise, if an application already has a SSA Logical disk open but since the open, it has been fenced out then calls to the read or write subroutine will fail, with errno set to **ENOCONNECT**.

The hardware fencing commands provide for a facility, to forcibly break through a fence. This command can be issued using the **SSADISK_ISALCMD** ioctl operation but it is necessary first to open the disk. A disk from which the host has been excluded can be opened by using the **openx** subroutine and specifying the **SSADISK_FENCEMODE** extension flag as described in SSA disk device driver device-dependent subroutines. While open in this mode no read or write operations will be permitted.

If a host has been excluded from access to a disk using fencing but that disk is also reserved to another host the reservation takes precedence. The return code from the **open** subroutine will be -1 and the global variable **errno** will be set to **EBUSY**. If the host attempts to break through the reservation by passing the *ext* parameter **SSADISK_FORCED_OPEN** to the **openx** subroutine, the reservation will be broken but the open will fail with **errno** set to **ENOCONNECT.** In order to break through the fence, the SSA logical disk must be opened in **SSADISK_FENCEMODE** and the **SSADISK_ISALCMD** ioctl operation used to issue the appropriate hardware command to break the fence condition.

Related Information

The SSA Adapter Device Driver, **ssadisk** SSA Disk Device Driver, SSA Subsystem Overview.

SSA Target Mode

The SSA Target-Mode interface (TMSSA) provides node-to-node communication through the SSA interface. The interface uses two special files that provide a logical connection to another node. One of the special files (the initiator-mode device) is used for write operations; the other (the target-mode device) is used for read operations. Data that is sent to a node is written to the initiator. Data that is read from a node is read from the target. The special files are:

/dev/tmssaXX.im
The initiator-mode device, which has an even, minor device number, and is write only.

/dev/tmssaXX.tm
The target-mode device, which has an odd, minor device number, and is read only.

The device is **tmssa*XX***, where ***XX*** is the node number of the using system with which these files communicate. You are not aware of which path connects the two nodes. The path can change if, for example, SSA loops are changed, nodes are switched off, or any other physical changes is made to the connected SSA loops. The TMSSA device driver can use any available path to the other node, but does not tell you which path is being used. Each node must have in its device configuration database a unique node number that is defined by the **node_number** attribute of the ssar device.

For example, in Node-to-Node Communication configuration, tmssa is, at first, using adapter ssa0 on node 1 and adapter ssa5 on node 2. Suddenly, the link between the adapters fails. The tmssa device driver automatically switches to using adapters ssa1 and ssa3 or adapters ssa1 and ssa4. The connections between nodes can be modified while they are in use, and the target-mode interface tries to recover.

The TMSSA uses either of two methods to read and write data:

- The blocking method, which waits until the I/O is complete or an error occurs before it returns control to you.
- The nonblocking method, which returns control to you immediately. With this method, the write operation occurs at a later time. The read, operation returns the amount of data that is available at the time of the operation. The amount of returned data is not necessarily the same as the amount that you requested.

The TMSSA device driver provides support for multiple concurrent read and write operations for different devices. It does not provide support for multiple read or write operations on the same device. The device driver blocks the operation until the device is free. Read and write operations can run concurrently on a particular device.

If a working path exists between two nodes, communication works. The path must be stable long enough for the driver to transmit the data. The maximum time taken to fail a write operation is (A * R * T), where A is the number of adapters in the host, R is the number of retries as defined by TM_MAXRETRY in the **/usr/include/sys/tmscsi.h** file, and T is the time-out before each retry. The minimum time taken to fail a write operation is the write time-out period. You can adjust the write time-out period and the retry-time out period; see ″TMCHGIMPARM (Change Parameters) tmssa Device Driver ioctl Operation″.

You can use the select and poll routines to check for read and write capability and can also be notified of a read or write being possible.

The amount of data that can be sent by one write operation in blocking mode has no limit, but the driver and adapter interface has been optimized for transfers of 512 bytes or less. In nonblocking mode, enough buffer space must be available for the write operation.

Each separate write is treated separately by the target, so, when reading, each separate write requires a separate read.

Configuring the SSA Target Mode

Each using system requires its own unique node number. The SSA adapter software specifies this node number, which is used by Target Mode SSA. The configuration database contains the ssar device. The **node_number** attribute sets the number for the node. Failure to have unique node numbers in the SSA loops causes unpredictable results with the target-mode interface. Node numbers that are not unique cause error logs. You can use the **ssavfynn** command to check for duplicate node numbers.

When the node is configured, it automatically inspects the existing SSA loops. It detects all nodes that are using the target mode SSA interface now. Each detected node is then added to the configuration database, if it is not already part of it. For each node that is added, **tmssa*XX*** is created, where ***XX*** is the node number of the detected node.

When configuration is complete, special files exist in the /dev directory. These files allow you to use the target mode interface with each node that is defined in the configuration database. Configuration does not need communication to be actually possible between the relevant using systems. Communication is needed only for the write operation.

Buffer Management

You can set the buffer sizes that are used by each device:

- To set the transmit buffer sizes, use the **chdev** command to adjust the **XmitBuffers** and **XmitBufferSize** attributes in the configuration database.
- To set the receive buffer size, use the **chdev** command to adjust the **RecvBuffers** and **RecvBufferSize** attributes in the configuration database.

The buffer sizes must be multiples of 128 bytes. The maximum buffer size is 512 bytes. A device can have as many buffers as it needs.

Data can be written into the buffers for the initiator-mode device at any time, even if nonblocked write operations are also transferring data from these buffers. The buffers for the target-mode device can be read at any time, even if a write operation to those buffers is occurring at the same time. It is not important if the sizes of the initiator-mode device buffers are different from the sizes of the target-mode device buffers to which the data is being sent. The total buffer space for the target-mode device, however, must be equal to, or greater than, the size of the initiator-mode device buffer size.

The SSA interface for target-mode transfers has been tuned for 512-byte transfers. Each write operation can send as much data as is required, unless that write operation is nonblocking. In a nonblocking write operation, the data being that is being written must be completely transferred to the device buffers. Therefore, the maximum amount of data that can be written during a nonblocked write operation is determined by the size of the device buffers.

Understanding Target-Mode Data Pacing

An initiator-mode device can send data faster than the associated target-mode device application can read it. This condition occurs when:

- The previous write operation is complete, but all the device buffers are in use, and no space is available for the next write operation.
- The write operation is not yet completed, and the device has no available buffers.

In both these instances, the target-mode device driver stops the write operation temporarily, and uses the retry mechanism to try again later. These actions can cause the write operation to fail. As a result, the initiator-mode device is unable to send any data to the target-mode device for the whole of the retry period. Alternatively, the write operation might time out.

Think about these possibilities when you set the buffer sizes and the number of buffers for the devices. Determine carefully the retry period, total write time-out period, and the amount of data that is being sent. For example, to write 64 KB of data with no retry operations, you need 64 KB read and write buffers. If you allow one retry operation, you need only 32 KB buffers.

Using SSA Target Mode

SSA Target Mode does not attempt to manage the data transfer between devices. It does, however,take action if buffers become full, and it ensures that read operations can read data from one write operation only. Any protocol that is needed to manage the communication of data must be implemented in user-supplied programs. The only delays that can occur when data is being received are delays that are characteristics of the SSA system and of the environment in which it operates, and delays that are caused by full buffers.

SSA Target Mode can concurrently send data to, and receive data from, all attached nodes. Blocking read and write operations do nothing until data is available to be read, or until the write operation is complete.

Execution of Target Mode Requests

The write operation transfers the data into the device buffers. When a buffer is full, the SSA adapter starts to transfer the data to the remote using system. At the same time, the user's application program continues to fill the device buffer with the remaining data that is being transferred. If the amount of data that is being written is larger than the available buffer space, the application program waits until more space becomes available in the device buffers. As each buffer is sent, the TMSSA device driver checks whether any more data is to be sent. If more data is to be sent, the device driver continues to send that data. If no more data is to be sent, and the write operation is in blocking mode, the device driver starts the waiting application program. If the write operation is in nonblocking mode, the write status is updated. If an unrecoverable error occurs, the write operation is ended, and the remaining buffers are discarded.

The read operation transfers received data from the device buffers to your application program. When the read operation ends, or the write operation stops sending data, the read operation returns the number of bytes read.

SSA tmssa Device Driver

Purpose

To provide support for using-system to using-system communications through the SSA target-mode device driver.

Syntax

```
#include </usr/include/sys/devinfo.h>

#include </usr/include/sys/tmscsi.h>

#include </usr/include/sys/scsi.h>

#include </usr/include/sys/tmssa.h>
```

Description

The Serial Storage Architecture (SSA) target-mode device driver provides an interface to allow using-system to using-system data transfer by using an SSA interface.

You can access the data transfer functions through character special files that are named **dev/tmssa*nn.xx***, where ***nn*** is the node number of the node with which you are communicating. The ***xx*** can be either **im** (initiator-mode interface), or **tm** (target-mode interface). The caller uses the initiator-mode to transmit data, and the target-mode interface to receive data.

When the caller opens the initiator-mode special file, a logical path is set up. This path allows data to be transmitted. The user-mode caller issues a **write**, **writev**, **writex**, or **writevx** system call to start sending data. The kernel-mode user issues an **fp_write** or **fp_rwuio** service call to start sending data. The SSA target-mode device driver then builds a **send** command to describe the transfer, and the data is sent to the device. The data can be sent as a blocking write operation, or as a nonblocking write operation. When the write entry point returns, the calling program can access the transmit buffer.

When the caller opens the target-mode special file, a logical path is set up. This path allows data to be received. The user-mode caller issues a **read**, **readv**, **readx**, or **readvx** system call to start receiving data. The kernel-mode caller issues an **fp_read** or **fp_rwuio** service call to start receiving data. The SSA target-mode device driver then returns data that has been received for the application program.

The SSA target mode device driver allows an initiator-mode device to get access to the data transfer functions through the write entry point; it allows a target-mode device to get access through the read entry point.

The only rules that the SSA target mode device driver observes to manage the sending and receiving of data are:

- Separate write operations need separate read operations.
- Receive buffers that are full, delay the send operation when it tries to resend after a delay.

The calling program must observe any other rules that are needed to maintain, or otherwise manage, the communication of data. Delays that occur when data is received or sent through the target mode device driver are that are characteristics of the hardware and software driver environment.

Configuration Information

When **tmssa*n*** is configured (where ***n*** is the remote node number), the **tmssan.im** and **tmssan.tm** special files are both created. An initiator-mode pair, or a target-mode pair, must exist for each device, whether either or both modes are being used. The target-mode node number for an attached device must be the same as the initiator-mode node number.

Each time that you use the **cfgmgr** command to configure the node, the target-mode device driver finds the remote nodes that are already connected, and automatically configures them. Each node is expected to be identified by a unique node number.

The target-mode device driver configuration entry point must be called only for the initiator-mode device number. The device driver configuration routine automatically creates the configuration data for the target-mode device minor number. This data is related to the initiator-mode data.

Device-Dependent Subroutines

The target-mode device driver provides support for the following subroutines:

- **open**
- **close**
- **read**
- **write**
- **ioctl**
- **select**

open Subroutine

The **open** subroutine allocates and initializes target, or initiator, device-dependent structures. No commands are sent to the device as a result of running the **open** subroutine.

The initiator-mode device or target-mode device must be configured but not already opened for that mode; otherwise, the **open** subroutine does not work. Before the initiator-mode device can be successfully opened, its special file must be opened for write operations only. Before the target-mode device can be successfully opened, its special file must be opened for read operations only.

Possible return values for the **errno** global variable include:

Value	Description
EBUSY	Attempted to run an **open** subroutine for a device instance that is already open.
EINVAL	Attempted to run an **open** subroutine for a device instance, but either a wrong open flag was used, or the device is not yet configured.
EIO	An I/O error occurred.
ENOMEM	The SSA device does not have enough memory resources.

close Subroutine

The **close** subroutine deallocates resources that are local to the target device driver for the target or initiator device. No commands are sent to the device as a result of running the **close** subroutine.

Possible return values for the **errno** global variable include:

Value	Description
EINVAL	Attempted to run a **close** subroutine for a device instance that is not configured or not opened.

Value	Description
EIO	An I/O error occurred.
EBUSY	The device is busy.

read Subroutine

Support for the **read** subroutine is provided only for the target-mode device. Support for data scattering is provided through the user-mode **readv** or **readvx** subroutine, or through the kernel-mode **fp_rwuio** service call. If the **read** subroutine is not successful, the return value is set to -1, and the **errno** global variable is set to the return value from the device driver. If the return value is something other than -1, the read operation was successful, and the return code indicates the number of bytes that were read. The caller should verify the number of bytes that were read. File offsets are not applicable and are ignored for target-mode read operations.

The adapter write operations provide the boundary that determines how read requests are controlled. If more data is received than is requested in the current read operation, the requested data is passed to the caller, and the remaining data is retained and returned for the next read operation for this target device. If less data is received in the **send** command than is requested, the received data is passed for the read request, and the return value indicates how many bytes were read.

If a write operation has not been completely received when a read request is made, the request blocks and waits for data. However, if the target device is opened with the O_NDELAY flag set, the read does not block; it returns immediately. If no data is available for the read request, the read is not successful, and the **errno** global variable is set to **EAGAIN**. If data is available, it is returned. The return value indicates the number of bytes that were received, whether the write operation for this data has ended or not.

Note: If the O_NDELAY flag is not set, the **read** subroutine can for an undefined time while it waits for data. Because, in a read operation, the data can come at any time, the device driver does not maintain an internal timer to interrupt the read. Therefore, if a time-out function is required, it must be started by the calling program.

If the calling program wants to break a blocked **read** subroutine, the program can generate a signal. The target-mode device driver receives the signal and ends the current **read** subroutine. If no bytes were read, the **errno** global variable is set to **EINTR**; otherwise, the return value indicates the amount of data that was read before the interrupt occurred. The read operation returns with whatever data has been received, whether the write operation has completed or not. If the remaining data for the write operation is received, it is put into a queue, where it waits for either another read request or a **close** command. When the target receives the signal and the current read is returned, another read operation can be started, or the target can be closed. If the read request that the calling program wants to break ends before the signal is generated, the read operation ends normally, and the signal is ignored.

The target-mode device driver attempts to queue received data in front of requests from the application program. A read-ahead buffer area is used to store the queued data. The length of this read-ahead buffer is determined by multiplying the value of the **RecvBufferSize** attribute by the value of the **RecvBuffers** attribute. These values are in the configuration database. While the application program runs **read** subroutines, the queued data is copied to the application data buffer, and the read-ahead buffer space is again made available for received data. If an error occurs while he data is being copied to the caller data buffer, the read operation fails, and the **errno** global variable is set to **EFAULT**. If the **read** subroutines are not run quickly enough to fill almost all the read-ahead buffers for the device, data reception is delayed until the application program runs a **read** subroutine again. When enough area is freed, data reception capability is restored from the device. Data might be delayed, but it is not lost or ignored.

The target-mode device driver controls only received data into its read entry point. The read entry point can optionally be used with the select entry point to provide a means of asynchronous notification of received data on one or more target devices.

Possible return values for the **errno** global variable include:

Value	Description
EAGAIN	Indicates that a nonblocking read request would have blocked, because data is available.
EFAULT	An error occurred while copying data to the caller buffer.
EINTR	Interrupted by a signal.
EINVAL	Attempted to run a read operation for a device instance that is not configured, not open, or is not a target-mode minor device number.
EIO	An I/O error occurred.

write subroutine

Support for the write entry point is provided only for the initiator-mode device driver. The write entry point generates one write operation in response to a calling program write request. If the device is opened with the O_NDELAY flag set, and the write request is for a length that is greater than the total buffer size of the device, the write request fails . The **errno** global variable is set to **EINVAL**. The total buffer size for the device is determined by multiplying the value of the **XmitBufferSize** attribute by the value of the **XmitBuffers** attribute. These values are in the configuration database.

Support for data gathering is through the user-mode **writev** or **writevx** subroutine, or through the kernel-mode **fp_rwuio** service call. The write buffers are gathered so that they are transferred, in sequence, as one write operation. The returned **errno** global variable is set to **EFAULT** if an error occurs while the caller data is being copied to the device buffers.

If the write operation is unsuccessful, the return value is set to -1 and the **errno** global variable is set to the value of the return value from the device driver. If the return value is other than -1, the write operation was successful and the return value indicates the number of bytes that were written. The caller should validate the number of bytes that are sent to check for any errors. Because the whole data transfer length is sent in a single write operation, you should suspect that a return code that is not equal to the expected total length is an error. File offsets are not applicable, and are ignored for target-mode write operations.

If the calling program needs to break a blocked write operation, a signal is generated. The target-mode device driver receives that signal, and ends the current write operation. The write operation that is in progress fails, and the **errno** global variable is set to **EINTR**. The write operation returns the number of bytes that were already sent, before the signal was generated. The calling program can then continue by issuing another write operation or an ioctl operation, or it can close the device. If the write operation that the caller attempts to break completes before the signal is generated, the write operation ends normally, and the signal is ignored.

If the buffers of remote using systems are full, or no device response status is received for the write operation, the target-mode device driver automatically retries the write operation. It retries the operation up to the number of times that is specified by the value **TM_MAXRETRY**. This value is defined in the **/usr/include/sys/tmscsi.h** file. By default, the target mode device driver delays each retry attempt by approximately two seconds to allow the target device to respond successfully. The caller can change the time delayed through the **TMCHGIMPARM** operation. If the write operation is still unsuccessful after the specified number of retries, it tries another SSA adapter. If this write operation has already tried all the SSA adapters, it fails. The calling program can retry the write operation, or perform other appropriate error recovery. No other error conditions are retried, but are returned with the appropriate **errno** global variable.

The target-mode device driver, by default, generates a time-out value, which is the amount of time allowed for the write operation to end. If the write operation does not end before the time-out value expires, the write operation fails. The time-out value is related to the length of the requested transfer, in bytes, and is calculated as follows:

```
timeout_value = ((transfer_length / 65536) + 1) * 20
```

In the calculation, 20 is the default scaling factor that generates the time-out value. The caller can customize the time-out value through the **TMCHGIMPARM** operation. The actual period that elapses

before a timeout occurs can be up to 10 seconds longer than the calculated value, because it is related to the operation of the hardware at the time of the write operation. A time-out value of zero means that no time-out occurs. A value of zero is not allowed when the write operation is nonblocking, because a deadlock might occur. Under this condition, **EINVAL** is returned for the write operation.

If the caller opened the initiator-mode device with the O_NDELAY flag set, the write operation is nonblocking. In this mode, the device checks whether enough buffer space is available for the write operation. If enough buffer space is not available, the write operation fails, and the **errno** global variable is set to **EAGAIN**. If enough buffer space is available, the write operation immediately ends with all the data written successfully. The write operation now occurs asynchronously. If you want to track the progress of this write operation, use the **TMIOSTAT** operation. The driver keeps the status of the last write operation, which is then reported by the **TMIOSTAT** operation.

Possible return values for the **errno** global variable include:

Value	Description
EFAULT	The write operation was unsuccessful because of a kernel service error. This value is applicable only during data gathering.
EINTR	Interrupted by signal.
EINVAL	Attempted to execute a write operation for a device instance that is not configured, not open, or is not an initiator-mode minor device number. If a nonblocked write operation, the transfer length is too long, or the time-out period is zero. If the transfer length is too long, try the operation again with a smaller transfer length. If the time-out period is zero, use **TMCHGIMPARM** to set the time-out value to another value.
EAGAIN	A nonblocked write operation could not proceed because not enough buffer space was available. Try the operation again later.
EIO	One of the following I/O errors occurred: • An error that cannot be produced again. • The number or retried operations reached the limit that is specified in **TM_MAXRETRY** without success on an error that cannot be reproduced. • The target-mode device of the remote node is not initialized or open. Do the appropriate error recovery routine.
ETIMEDOUT	The command has timed out. Do the appropriate error recovery routine.

ioctl Subroutine

The following ioctl operations are provided by the target-mode device driver. Some are specific to either the target-mode device or the initiator-mode device. All require the respective device instance be open for the operation run.

Operation	Description
IOCINFO	Returns a structure defined in the **/usr/include/sys/devinfo.h** file.
TMCHGIMPARM	Allows the caller to change some parameters that are used by the target mode device driver for a particular device instance.
TMIOSTAT	Allows the caller to get status information about the previously run write operation.

Possible return values for the **errno** global variable include:

Value	Description
EFAULT	The kernel service failed when it tried to access the caller buffers.
EINVAL	The device not open or not configured. The operation is not applicable to mode of this device. A parameter that is not valid was passed to the device driver.

select Entry Point

The select entry point allows the caller to know when a specified event has occurred on one or more target-mode devices. The event *input* parameter allows the caller to specify about which of one or more conditions it wants to be notified by a bitwise OR of one or more flags. The target-mode device driver provides support for the following select events:

Event	Description
POLLIN	Check whether received data is available.
POLLSYNC	Return only events that are currently pending. No asynchronous notification occurs.

The additional events, POLLOUT and POLLPRI, are not applicable. The target-mode device driver does not, therefore, provide support for them.

The *reventp* output parameter points to the result of the conditional checks. The device driver can return a bitwise OR of the following flags:

POLLIN	Received data is available.

The *chan* input parameter is used for specifying a channel number. This parameter is not applicable for nonmultiplexed device drivers. It should be set to 0 for the target-mode device driver.

The POLLIN event is indicated by the device driver when any data is received for this target instance. A nonblocking **read** subroutine, if subsequently issued by the caller, returns data. For a blocking read subroutine, the read does not return until either the requested length is received, or the write operation ends, whichever comes first.

Asynchronous notification of the POLLIN event occurs when received data is available. This notification occurs only if the select event POLLSYNC was not set.

The initiator-mode device driver provides support for the following select events:

Event	Description
POLLOUT	Check whether output is possible.
POLLPRI	Check whether an error occurred with the write operation.
POLLSYNC	Return only events that are currently pending. No asynchronousnotification occurs.

An additional event POLLIN is not applicable and has no support from the initiator-mode device driver.

The *reventp* output parameter points to the result of the conditional checks. The device driver can return a bitwise OR of the following flags:

Flag	Description
POLLOUT	If the initiator device is opened with the O_NDELAY flag, some buffer space is not being used now. Otherwise, this event is always set for the initiator-mode device.
POLLPRI	An error occurred with the latest write operation.

Asynchronous notification of the POLLOUT event occurs when buffer space is made available for further write operations.

Asynchronous notification of the POLLPRI event occurs if an error occurs with a write operation. Note that the error might be recovered successfully by the device driver.

Possible return values for the **errno** global variable include:

Value	Description
EINVAL	A specified event has no support, or the device instance is not configured or not open.

Errors

Errors that are detected by the target-mode device driver can be one of the following:

- A hardware error that occurred while receiving data, and cannot be reproduced
- A hardware error that occurred during an adapter command, and cannot be reproduced
- A hardware error that has not been recovered
- A software error that has been detected by the device driver

The target-mode device driver passes error-recovery responsibility for all detected errors to the caller. For these errors, the target-mode device driver does not know if this type of error is permanent or temporary. These types of errors are handled as temporary errors.

Only errors that the target-mode device driver can itself recover through retry operations can be determined to be either temporary or permanent. The error is ignored if it succeeds during retry (a recovered error). The return code to the caller indicates success if a recovered error occurs, or failure if an unrecovered error occurs. The caller can retry the command or operation, but success is probably low for unrecovered errors.

TMSSA does no error logging. If an error occurs, that error might be logged by the adapter device driver.

tmssa Special File

Purpose

To provide access to the SSA tmssa device driver.

Description

The Serial Storage Architecture (SSA) target-mode device driver provides an interface that allows the SSA interface to be used for data transfer from using system to using system.

You can access the data transfer functions through character special files that are named **dev/tmssa*nn.xx***, where ***nn*** is the node number of the node with which you are communicating. The ***xx*** can be either **im** (initiator-mode interface), or **tm** (target-mode interface). The caller uses the initiator-mode to transmit data, and the target-mode interface to receive data.

The least significant bit of the minor device number indicates to the device driver which mode interface is selected by the caller. When the least significant bit of the minor device number is set to 1, the target-mode interface is selected. When the least significant bit is set to 0, the initiator-mode interface is selected. For example, **tmssa1.im** should be defined as an even-numbered minor device number to select the initiator-mode interface. **tmssa1.tm** should be defined as an odd-numbered minor device number to select the target-mode interface.

When the caller opens the initiator-mode special file, a logical path is set up. This path allows data to be transmitted. The user-mode caller issues a **write**, **writev**, **writex**, or **writevx** system call to start data transmission. The kernel-mode user issues an **fp_write** or **fp_rwuio** service call to start data transmission. The SSA target-mode device driver then builds a **send** command to describe the transfer, and the data is sent to the device. The transfer can be done as a blocking write operation or as a nonblocking write operation. When the write entry point returns, the calling program can access the transmit buffer.

When the caller opens the target-mode special file, a logical path is set up. This path allows data to be received. The user-mode caller issues a **read**, **readv**, **readx**, or **readvx** system call to start the receiving of data. The kernel-mode caller issues an **fp_read** or **fp_rwuio** service call to start the receiving of data. The SSA target-mode device driver then returns data that was received for the application program.

Related Information

The **close** subroutine, **open** subroutine, **read** or **readx** subroutine, **write** or **writex subroutine**.

IOCINFO (Device Information) tmssa Device Driver ioctl Operation

Purpose

To return information about the device in a structure that is defined in the **/usr/include/sys/devinfo.h** file.

Description

This operation allows you to supply a pointer to the address of an area of type **struct devinfo** in the *arg* parameter to the **IOCINFO** operation. This structure is defined in the **/usr/include/sys/devinfo.h** file. The SCSI target-mode union is used for this as follows:

Initiator-Device

buf_size	Size of transmit buffer.
num_bufs	Number of transmit buffers.
max_transfer	Unused. Set to zero.
adap_devno	Major or Minor devno of SSA adapter to be used for the next transmit operation.

Use **TM_GetDevinfoNodeNum( )** to read the node number to which the data is sent.

Target-Device

buf_size	Size of receive buffer.
num_bufs	Number of receive buffers.
max_transfer	Unused. Set to zero.
adap_devno	Major or Minor devno of SSA adapter initially used by the paired initiator-mode device.

Use **TM_GetDevinfoNodeNum( )** to read the node number from which the data is received.

The remainder of the structure is filled as follows:

devtype	DD_TMSCSI.
flags	Set to zero.
devsubtype	DS_TM.

TMIOSTAT (Status) tmssa Device Driver ioctl Operation

Purpose

To allow the caller to put the status information for the current or previous write operation into a structure that is defined in the **/usr/include/sys/tmscsi.h** file.

Description

This operation returns information about the last write operation. Because a nonblocking write operation might still be running, you must ensure that the status information applies to a particular write operation. The tm_get_stat structure in the **/usr/include/sys/tmscsi.h** file is used to indicate the status, as follows:

status_validity
Bit 0 set, scsi_status valid

scsi_status

```
SC_BUSY_STATUS Write operation in progress
SC_GOOD_STATUS Write operation completed successfully
SC_CHECK_CONDITION Write operation failed
```

general_card_status
Unused. Set to zero.

b_error
errno for a failed write operation, or zero.

b_resid
Updated uio_resid for the write operation.

resvd1
Unused. Set to zero.

resvd2
Unused. Set to zero.

Note: The tm_get_stat structure works only for the initiator device.

TMCHGIMPARM (Change Parameters) tmssa Device Driver ioctl Operation

Purpose

To allow the caller to change the parameters that are used by the target-mode device driver.

Description

This operation allows the caller to change the default set up of the device. It is allowed only for the initiator-mode device. The *arg* parameter to the **TMCHGIMPARM** operation contains the address of the **tm_chg_im_parm** structure that is defined in the **/usr/include/sys/tmscsi.h** file.

Default values that are used by the device driver for these parameters usually do not require change. For some calling programs, however, default values can be changed to fine tune timing parameters that are related to error recovery.

When a parameter is changed, it remains changed until another **TMCHGIMPARM** operation occurs, or until the device is closed. When the device is opened, the parameters are set to the default values.

Parameters that can be changed with this operation are:

- The delay (in seconds) between device-driver-initiated retries of **send** commands
- The time allowed before the write operation times out.

To indicate which of the possible 0 parameters the caller is changing, the caller sets the appropriate bit in the chg_option field. The caller can change only the *retry* parameters, or only the *time out* parameters, or both types of parameter.

To change the delay between **send** command retries, the caller sets the TM_CHG_RETRY_DELAY flag in the chg_option field and puts the required delay value (in seconds) into the new_delay field of the structure. With this command, the retry delay can be changed with this command to any value 0 through 255, where 0 instructs the device driver to use as little delay as possible between retries. The default value is approximately two seconds.

To change the **send** command time-out value, the caller sets the TM_CHG_SEND_TIMEOUT flag in the chg_option field, sets the desired flag in the timeout_type field, and puts the desired time-out value into the new_timeout field of the structure. One flag must be set in the time_out field to indicate the required form of the timeout. If the TM_FIXED_TIMEOUT flag is set in the timeout_type field, the value that is put into the new_timeout field is a fixed time-out value for all **send** commands. If the TM_SCALED_TIMEOUT flag is set in the timeout_type field, the value that is put into the new_timeout field is a scaling-factor used in the calculation for timeouts as shown under the description of the write entry point. The default **send** command time-out value is a scaled time-out with a scaling factor of 10.

Regardless of the value of the timeout_type field, if the new_timeout field is set to a value of 0, the caller specifies “no time out” for the **send** command, allowing the command to take an indefinite amount of time. If the calling program wants to end a write operation, it generates a signal. This option is only allowed for nonblocking write operations.

Chapter 8. Serial DASD Subsystem

Serial DASD Subsystem Device Driver

Purpose

Supports the serial DASD physical volume (fixed-disk) subsystem device driver.

Syntax

```
#include <sys/devinfo.h>
#include <sys/scsi.h>
#include <sys/serdasd.h>
```

Description

The serial DASD Subsystem is a serial-link disk subsystem, consisting of the following components:

- Serial DASD (direct access storage device)
- Controllers
- Adapters

Note: The serial DASD subsystem adapters are not SCSI adapters, and the subsystem is not a SCSI Bus. The controllers and DASD subsystem use the SCSI protocol for command packaging. They are not SCSI devices.

Serial DASD

Attention: Potential for data corruption or system crashes: Data corruption, loss of data or loss of system integrity occurs if devices supporting paging, logical volumes, or mounted file systems are accessed using block special files. Block special files are provided for logical volumes and disk devices and are solely for system use in managing file systems, paging devices, and logical volumes. Direct access to physical disks through **/dev/hdisk***n* block special files could cause the system to experience performance difficulties. These difficulties include data consistency problems between data in the block I/O buffer cache and data in system pages.

Note: The maximum transfer request through the **/dev/hdisk***n* block special file is 128K bytes.

The **/dev/rhdisk** special file provides raw I/O access and control functions to the physical disk device drivers. Raw I/O access is provided through the **/dev/rhdisk0**, **/dev/rhdisk1**, and other character special files.

The prefix of **r** on a special file name indicates that the drive is accessed as a raw device rather than a block device. Performing raw I/O with a fixed disk requires that all data transfers be in multiples of the disk block size. Also, all **lseek** subroutines that are made to the raw-disk device driver must result in a file pointer value that is a multiple of the disk block size.

Controllers

The **/dev/serdasdc***n* special file provides access and control functions to the Serial DASD Subsystem controllers. Access to the controllers, primarily for diagnostic purposes, is through the **/dev/serdasdc0**, **/dev/serdasdc1**, and other special files.

Adapters

The **/dev/serdasda***n* special file provides access and control functions to the Serial DASD Subsystem adapters. Access to the adapters, primarily for diagnostic purposes, is through the **/dev/serdasda0**, **/dev/serdasda1**, and other special files.

There is also a **/dev/dserdasda***n* adapter special file for each adapter reserved for use by the RAS configuration daemon.

Related Information

Device-Dependent Subroutines for the Serial DASD Subsystem.

Error Conditions for Serial DASD Subroutines .

Reliability, Availability, and Serviceability (RAS) Daemon for the Serial DASD Subsystem .

The **/dev/rhdisk** special file, **/dev/serdasda** special file, **/dev/serdasdc** special file.

The **lseek** subroutine.

Device-Dependent Subroutines for the Serial DASD Subsystem

There are three classes of device dependent subroutines for the serial DASD subsystem:

- Device-dependent subroutines for serial DASD operations
- Device-dependent subroutines for serial DASD controller operations
- Device-dependent subroutines for serial DASD adapter operations

Related Information

Error Conditions for Serial DASD Subroutines.

Reliability, Availability, and Serviceability (RAS) Daemon for the Serial DASD Subsystem.

Serial Direct Access Storage Device (DASD) Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Serial DASD Subsystem Device Driver.

Device-Dependent Subroutines for Serial DASD Operations

Use the **open**, **openx**, **close**, **read**, **readx**, **write**, **writex**, and **ioctl** subroutines to implement Direct Access Storage Device (DASD) operations. Observe the special considerations for using these subroutines:

- **openx**
- **readx**
- **writex**
- **ioctl**

openx Subroutine

The **openx** subroutine is intended primarily for use by diagnostic commands and utilities. Appropriate authority is required for execution. Attempting to execute this subroutine without the proper authority results in a return value of -1, with the **errno** global variable set to **EACCES**.

The *ext* parameter passed to the **openx** subroutine selects the operation to be used for the target DASD. The **/usr/include/sys/scsi.h** file defines possible values for the *ext* parameter. The parameter can contain any combination of the following flag values logically ORed together:

Value	Description
SC_DIAGNOSTIC	Places the selected DASD in Diagnostic mode. This mode is singularly entrant. When a DASD is in Diagnostic mode, no Serial DASD subsystem operations are performed during **open** or **close** operations, and error logging is disabled. In Diagnostic mode only the **close** and **ioctl** operations are accepted. All other DASD-supported subroutines return a value of -1, and the **errno** global variable is set to **EACCES.**
SC_FORCED_OPEN	Forces a DASD reset regardless of whether another initiator has the DASD reserved, thereby overtaking the reservation. The DASD reset is sent to the DASD before the **open** sequence begins. Otherwise, the **open** sequence executes normally.
SC_RETAIN_RESERVATION	Retains the reservation of the DASD after a **close** operation by not issuing the release. This flag prevents other initiators from using the DASD unless they break the host machine's reservation.
SD_NO_RESERVE	Prevents the reservation of a DASD during an **openx** subroutine call to a DASD. This operation is provided so a DASD can be controlled by two processors that synchronize their activity by their own software means. This is defined in the **/usr/include/sys/serdasd.h** file as the **SC_NO_RESERVE** flag, which is defined in the **/usr/include/sys/scsi.h** file.

readx and writex Subroutines

The **readx** and **writex** subroutines provide additional parameters affecting the raw data transfer. These subroutines pass the *ext* parameter, which specifies request options. Construct these options by using logical OR with any of the following values:

Value	Description
WRITEV	Indicates a request for write verification.
HWRELOC	Indicates a request for hardware relocation (safe relocation only).
UNSAFEREL	Indicates a request for unsafe hardware relocation.

ioctl Subroutine

The **ioctl** subroutine can call the **IOCINFO**, **SD_SCSICMD**, **SD_RESET**, **SD_SET_FENCE**, **SD_CLEAR_FENCE**, **DD_CONC_REGISTER**, and **DD_CONC_UNREGISTER** operations.

IOCINFO

The **IOCINFO** operation is the only operation defined for all device drivers that use the **ioctl** subroutine. The **IOCINFO** operation returns the **devinfo** structure defined in the **/usr/include/sys/devinfo.h** file. This ioctl operation can be directed to an adapter, controller, or DASD. The device can be opened in normal mode for the **ioctl** subroutine. Information for DASD is returned through the **scdk** structure within the **devinfo** structure.

SD_SCSICMD

When the device has been successfully opened in Diagnostic mode, the **SD_SCSICMD** operation provides the means for issuing any Serial DASD subsystem command to a specified device. The Serial DASD subsystem commands are modeled after those for SCSI. The following Serial DASD subsystem commands are valid and use the same command descriptor block, including the operation code, as their corresponding SCSI command. The following SCSI commands are defined in the **/usr/include/sys/scsi.h** file:

- Format Unit
- Inquiry
- Mode Select
- Mode Sense

- Read (6)
- Read (10)
- Read Capacity
- Reassign Blocks
- Release
- Request Sense
- Reserve
- Start Stop Unit
- Test Unit Ready
- Verify
- Write
- Write And Verify
- Write Buffer
- Write Sense

The following Serial DASD subsystem commands can be issued when the device is not in Diagnostic mode, but the caller has root authority:

- Fence
- Inquiry
- Read
- Read Extended
- Receive Diagnostics
- Request Sense
- Write Buffer

The Serial DASD Fence command does not correspond with any SCSI command. For more information, see Serial DASD Fence Command .

If the **SD_SCSICMD** operation is issued with any other Serial DASD subsystem command and the device is not in Diagnostic mode, the **SD_SCSICMD** operation returns a value of -1 and sets the **errno** global variable to a value of **EACCES**. The device driver performs no error recovery or error logging when this ioctl operation fails. The **status_validity** byte, **scsi_bus_status** byte, and the **adapter_status** byte are returned via the *arg* parameter. This parameter contains the address of a **sc_iocmd** structure, which is defined in the **/usr/include/sys/scsi.h** file.

The **devinfo** structure, which is returned from the **IOCINFO** ioctl operation, defines the maximum transfer size for the command. The structure returns a value of -1 and sets the **errno** global variable to a value of **EINVAL** if an attempt is made to transfer more than the maximum transfer size. If the Serial DASD subsystem command cannot complete in the time specified in the **sc_iocmd** structure, a -1 is returned, and the **errno** global variable is set to a value of **ETIMEDOUT**.

The **SD_SCSICMD** operation uses the **sc_iocmd** structure with the following status validity values:

Value	Description
0x00	Command successful
0x01	Valid **scsi_bus_status** byte only
0x02	Valid adapter status only
0x04	Valid alert register contents

Note: Except when the adapter status is **SC_ADAPTER_HDW_FAILURE**, if the adapter status is valid, then the alert register contents are also valid. When the device driver fails a command due to internal error recovery without communicating with the adapter, the **SC_ADAPTER_HDW_FAILURE** adapter status is returned. For all other adapter status values, the adapter status is defined as part of the general card status in the **/usr/include/sys/scsi.h** file.

The **SD_SCSICMD** operation also uses the following reserved fields:

Field	Description
resvd1	Returned as the controller status byte of the alert register. The possible values for this field are defined in the ″Controller Status Byte Codes″ section of the **/usr/include/sys/serdasd.h** file.
resvd2	Returned as the adapter status byte of the alert register. The possible values for this field are defined in the ″Adapter Status Byte Codes″ section of the **/usr/include/sys/serdasd.h** file.
resvd6	Manipulates the queue control and ordering of this command. The caller must set this field to one of the following values: **0x00** None, unqueued **0x40** Invalid **0x80** Ordered **0xC0** Unordered
resvd7	Specifies an extension to the Serial DASD subsystem command. The caller must set this field to one of the following values: **0x00** No extension **0x20** Split write enabled **0x10** Split read enabled

SD_RESET

The **SD_RESET** ioctl provides an interface for issuing resets and quiesces to the DASD. The DASD is the **/dev/hdisk***N* file, where *N* is 0 or a positive integer. To send a reset or quiesce to a DASD, the DASD must be opened in **SC_DIAGNOSTIC** mode. The *arg* parameter of the **ioctl** call is a pointer to a **sd_ioctl_parms** structure, which is defined in the **/usr/include/sys/serdasd.h** file.

If the **SD_RESET** ioctl is issued when the device is not in Diagnostic mode, the ioctl returns a value of -1 and sets the **errno** global variable to a value of **EACCES**. If the Serial DASD adapter does not respond to the **SD_RESET** ioctl, a value of -1 is return, and the **errno** global variable is set to a value of **EIO**.

The following fields of the **sd_ioctl_parms** structure are used with this ioctl command:

Field	Description
reset_type	Contains one of the following values:
SD_RESET_OP	Indicates a full reset
	SD_QUIESCE_OP Indicates a quiesce
status_validity	Indicates either successful completion or the status of the adapter or controller. This field may have one of the following values: **0x00** Command successful **0x01** Valid adapter status **0x02** Valid controller status **0x04** Valid device driver status only

Note: If the timeout field of the **sc_iocmd** structure is set to 0, the calling process is responsible for handling timeouts. In this situation, the device driver will never time out the command.

If a **SD_RESET** ioctl is issued when the device is not in Diagnostic mode, the ioctl returns a value of -1 and sets the **errno** global variable to a value of **EACCES**.

SD_SET_FENCE

The **SD_SET_FENCE** ioctl provides an interface for establishing a fence for Serial DASD. The fence is established via the Mask and Swap Fence command with the Force Fence bit off. The DASD is the **/dev/hdisk***N* file, where *N* is 0 or a positive integer. The *arg* parameter of the **ioctl** call is a pointer to a **sd_ioctl_parms** structure, which is defined in the **/usr/include/sys/serdasd.h** file.

The **SD_SET_FENCE** ioctl uses the following fields of the **sd_ioctl_parms** structure:

Field	Description
`resvd1`	Indicates that the caller sets the 16-bit fence mask used by the Mask and Swap fence command. The fence mask specifies which bits in the fence register to change to the value specified in the fence data. The bit can have one of the following values: **1** Indicates the corresponding fence register bit should be changed to the value of the bit specified in the fence data. **0** Indicates the corresponding fence register bit should not be changed.
`resvd2`	Indicates that the caller sets the 16-bit fence data used by the Mask and Swap fence command. The bits in the fence data specify which hosts are fenced out. The bit can have one of the following values: **1** Indicates the corresponding host connection on the back of the controller is fenced out. **0** Indicates the corresponding host connection is not fenced out.
`status_validity`	Indicates either the successful completion of the fence command or the existence of adapter or controller status.
`adapter_status`	Indicates the adapter status, if valid, upon completion of the Fence command.
`resvd3`	Contains the fence position indicator returned by the Fence command.
`resvd4`	Contains the old fence value returned by the Fence command.
`resvd5`	Contains the current fence for the DASD.

If the **SD_SET_FENCE** ioctl is successful, the Serial DASD device driver reissues the Fence anytime it detects a power off and subsequent power on of a Serial DASD.

Note: If successive **SD_SET_FENCE** ioctls are issued without any **SD_CLEAR_FENCE** ioctls in between, the Serial DASD maintains the fence as a composite of all the uncleared **SD_SET_FENCE** ioctls.

If the Serial DASD does not support fencing, the **SD_SET_FENCE** ioctl returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. If the `resvd1` and `resvd2` fields of the **sd_ioctl_parms** structure would fence out this host, this ioctl would return a value of -1 and sets the **errno** global variable to a value of **EINVAL**.

For more information on the Fence command, see ″Serial DASD Fence Command″ .

SD_CLEAR_FENCE

The **SD_CLEAR_FENCE** ioctl provides an interface for removing the fence established by the **SD_SET_FENCE** ioctl for a DASD. The DASD is the **/dev/hdisk***N* file, where *N* is 0 or a positive integer. The DASD can be opened in normal mode for the **ioctl** subroutine. The *arg* parameter of the **ioctl** call is a pointer to a **sd_ioctl_parms** structure, which is defined in the **/usr/include/sys/serdasd.h** file.

The **SD_CLEAR_FENCE** ioctl uses the following fields:

Field	Description
`status_validity`	Indicates either the successful completion of the Fence command or the existence of adapter or controller status.

Field	Description
adapter_status	Indicates the adapter status, if valid, upon completion of the Fence command.
controller_status	Indicates the controller status, if valid, upon completion of the Fence command.
resvd3	Contains the fence position indicator returned by the Fence command.
resvd4	Contains the old fence value returned by the Fence command.

If a fence has not been established with the **SD_SET_FENCE** ioctl, the **SD_CLEAR_FENCE** ioctl returns a value of -1 and sets the **errno** global variable to a value of **EINVAL**. If the device does not support fencing, a value of -1 is returned, and the **errno** global variable is set to a value of **EINVAL**.

Note: If successive **SD_SET_FENCE** ioctls are issued without any **SD_CLEAR_FENCE** ioctls in between, the Serial DASD maintains the fence as a composite of all the uncleared **SD_SET_FENCE** ioctls. The **SD_CLEAR_FENCE** ioctl clears the composite fence.

DD_CONC_REGISTER

The **DD_CONC_REGISTER** ioctl provides an interface for registering one kernel extension with the Serial DASD device driver for a given DASD. The DASD is the **/dev/hdisk***N* file, where *N* is 0 or a positive integer. Registration is needed to enable the concurrent mode of operation for Serial DASDS that are shared between hosts. The *arg* parameter for this ioctl is a pointer to a **dd_conc_register** structure, which is defined in the **/usr/include/sys/ddconc.h** file. The kernel extension calling this ioctl must set the conc_intr_addr field of this structure to its special interrupt handler for processing concurrent mode commands. See ″Serial DASD Concurrent Mode of Operation Interface″ for more information.

If the ioctl returns with a value of 0, the conc_func_addr field of the **dd_conc_register** structure contains the Serial DASD device driver's special entry point for issuing concurrent mode commands. A value of -1 is returned, and the **errno** global variable is set to a value of **EINVAL** for the following occurrences:

- The ioctl is not called from a kernel extension.
- A kernel extension has already registered for this DASD.
- The Serial DASD does not support the concurrent mode operation.

DD_CONC_UNREGISTER

The **DD_CONC_REGISTER** ioctl provides an interface for unregistering one kernel extension with the SERIAL DASD device driver for a given DASD. The DASD is the **/dev/hdisk***N* file, where *N* is 0 or a positive integer.

A value of -1 is returned, and the **errno** global variable is set to a value of **EINVAL** for the following occurrences:

- A kernel extension is not registered.
- The **DD_CONC_UNREGISTER** ioctl is not called from a kernel extension.

Related Information

The **close** subroutine, **ioctl** subroutine, **open** or **openx** subroutine, **readx** subroutine, **writex** subroutine.

Device-Dependent Subroutines for Serial DASD Controller Operations.

Device-Dependent Subroutines for Serial DASD Adapter Operations.

Error Conditions for Serial DASD Subroutines.

Reliability, Availability, and Serviceability (RAS) Daemon for the Serial DASD Subsystem.

Serial Direct Access Storage Device (DASD) Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Device-Dependent Subroutines for Serial DASD Controller Operations

Use the **open**, **openx**, **close**, and **ioctl** subroutines to implement Direct Access Storage Device (DASD) operations. Observe special considerations for using these subroutines:

- **openx**
- **ioctl**

open and close Subroutines

The **openx** subroutine is used primarily by diagnostic commands and utilities. Users must have root authority to execute this subroutine. Attempting to execute this subroutine without the proper authority results in a return value of -1, with the **errno** global variable set to **EACCES**.

The *ext* parameter selects the operation to be used for the target controller. The **/usr/include/sys/scsi.h** file defines the **SC_DIAGNOSTIC** value for the *ext* parameter.

The **SC_DIAGNOSTIC** extended parameter places the selected controller in Diagnostic mode. This mode is singularly entrant. When a controller is in Diagnostic mode, all DASDs on that controller are placed in Diagnostic mode, and error logging is disabled.

Note: If a controller is opened in Diagnostic mode, commands can be addressed to that controller or any DASD on that controller.

ioctl Subroutine

The **ioctl** subroutine can call the **IOCINFO**, **SD_SCSICMD**, and **SD_RESET** operations.

IOCINFO

The **IOCINFO** operation is the only operation defined for all device drivers that use the **ioctl** subroutine. The **IOCINFO** operation returns the **devinfo** structure defined in the **/usr/include/sys/devinfo.h** file. This ioctl operation can be directed to an adapter, controller, or DASD. The device can be opened in normal mode for this ioctl operation.

SD_SCSICMD

When the device has been successfully opened in Diagnostic mode, the **SD_SCSICMD** operation provides the means for issuing any Serial DASD subsystem command to a specified device. The Serial DASD subsystem commands are modeled after those for SCSI. The following Serial DASD subsystem commands are valid and use the same command descriptor block, including the operation code, as their corresponding SCSI command. The following SCSI commands are defined in the **/usr/include/sys/scsi.h** file:

- Format Unit
- Inquiry
- Mode Select
- Mode Sense
- Read (6)
- Read (10)
- Read Capacity
- Reassign Blocks
- Release
- Request Sense
- Reserve
- Start Stop Unit

- Test Unit Ready
- Verify
- Write
- Write And Verify
- Write Buffer
- Write Sense

The following Serial DASD subsystem commands can be issued when the device is not in Diagnostic mode, but the caller has root authority:

- Fence
- Inquiry
- Read
- Read Extended
- Receive Diagnostics
- Request Sense
- Write Buffer

The Serial DASD Fence command does not correspond with any SCSI command. For more information, see ″Serial DASD Fence Command″ .

If the **SD_SCSICMD** operation is issued with any other Serial DASD subsystem command and the device is not in Diagnostic mode, the **SD_SCSICMD** operation returns a value of -1 and sets the **errno** global variable to a value of **EACCES**. The device driver performs no error recovery or error logging when this ioctl operation fails. The **status_validity** byte, **scsi_bus_status** byte, and the **adapter_status** byte are returned via the *arg* parameter. This parameter contains the address of a **sc_iocmd** structure, which is defined in the **/usr/include/sys/scsi.h** file.

The **devinfo** structure, which is returned from the **IOCINFO** ioctl operation, defines the maximum transfer size for the command. The structure returns a value of -1 and sets the **errno** global variable to a value of **EINVAL** if an attempt is made to transfer more than the maximum transfer size. If the Serial DASD subsystem command cannot complete in the time specified in the **sc_iocmd** structure, a -1 value is returned, and the **errno** global variable is set to a value of **ETIMEDOUT**.

The **SD_SCSICMD** operation uses the **sc_iocmd** structure with the following status validity values:

Value	Description
0x00	Command successful
0x01	Valid **scsi_bus_status** byte only
0x02	Valid adapter status only
0x04	Valid alert register contents

Note: Except when the adapter status is **SC_ADAPTER_HDW_FAILURE**, if the adapter status is valid, the alert register contents are also valid. When the device driver fails a command due to internal error recovery without communicating with the adapter, the **SC_ADAPTER_HDW_FAILURE** adapter status is returned. For all other adapter status values, the adapter status is defined as part of the general card status in the **/usr/include/sys/scsi.h** file.

The **SD_SCSICMD** operation uses the **sc_iocmd** structure using the following reserved fields:

Field	Description
`resvd1`	Returned as the controller status byte of the alert register. The possible values for this field are defined in the ″Controller Status Byte Codes″ section of the **/usr/include/sys/serdasd.h** file.

Field	Description
resvd2	Returned as the adapter status byte of the alert register. The possible values for this field are defined in the ″Adapter Status Byte Codes″ section of the **/usr/include/sys/serdasd.h** file.
resvd5	Specifies the address of the device. The address is comprised of eight bits:
4	Direction (0 = DASD, 1 = Controller)
	0-3 DASD (LUN)
resvd6	Manipulates the queue control and ordering of the **SD_SCSICMD** command. The caller must set this field to one of the following values:
	0x00 None, unqueued
	0x40 Invalid
	0x80 Ordered
	0xC0 Unordered
resvd7	Specifies an extension to the Serial DASD subsystem command. The caller must set this field to one of the following values:
	0x00 No extension
	0x20 Split write enabled
	0x10 Split read enabled

Note: If the timeout field of the **sc_iocmd** structure is set to 0, the calling process is responsible for handling timeouts. The device driver will never time out the command if the value of the structure is 0.

SD_RESET

The **SD_RESET** operation provides an interface for issuing resets and quiesces to the Serial DASD controller or its storage device. The controller is the **/dev/serdasdc***N* file, where *N* is 0 or a positive integer. To send a reset or quiesce to a controller, the controller must be opened in **SC_DIAGNOSTIC** mode. To send a reset or quiesce to a controller's DASD, the controller can be opened with root authority in either **SC_DIAGNOSTIC** mode or normal mode. The *arg* parameter of the **ioctl** call is a pointer to a **sd_ioctl_parms** structure, which is defined in the **/usr/include/sys/serdasd.h** file.

The following fields of the **sd_ioctl_parms** structure are used with this ioctl command.

Field	Description
reset_type	Contains one of the following values:
	SD_RESET_OP Indicates a full reset.
	SD_QUIESCE_OP Indicates a quiesce.
	SD_DASD_RESET Indicates the controller's DASD is to be reset. The controller's DASD is specified in the resvd1 field.
status_validity	Indicates either successful completion or the status of the adapter or controller. This field may have one of the following values:
	0x00 Command successful
	0x01 Valid adapter status
	0x02 Valid controller status
	0x04 Valid device driver status only
adapter_status	Indicates the adapter status, if valid, upon completion of the reset operation. The possible values for this field are defined in the ″Adapter Status Byte Codes″ section of the **/usr/include/sys/serdasd.h** file.

Field	Description
controller_status	Indicates the controller status, if valid, upon completion of the reset operation. The possible values for this field are defined in the ″Controller Status Byte Codes″ section of the **/usr/include/sys/serdasd.h** file.
resvd1	Specifies the logical unit number (LUN) of the DASD to be reset if the reset type is **SD_DASD_RESET**.

If the **SD_RESET** operation with a **SD_RESET_OP** or **SD_QUIESCE** reset type is issued when the device is not in Diagnostic mode, the operation returns a value of -1 and sets the **errno** global variable to value of **EACCES**.

Related Information

The **ioctl** subroutine, **open**, **openx** subroutine.

Serial DASD Subsystem Device Driver.

Device-Dependent Subroutines for Serial DASD Operations

Device-Dependent Subroutines for Serial DASD Adapter Operations.

Error Conditions for Serial DASD Subroutines.

Reliability, Availability, and Serviceability (RAS) Daemon for the Serial DASD Subsystem.

Serial DASD Fence Command.

Serial DASD Concurrent Mode of Operation Interface.

Serial Direct Access Storage Device (DASD) Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Device-Dependent Subroutines for Serial DASD Adapter Operations

Use the **open**, **openx**, **close**, **read**, **readx**, **write**, **writex**, and **ioctl** subroutines to implement Direct Access Storage Device (DASD) operations. Observe the special considerations for using these subroutines:

- **openx**
- **ioctl**

openx Subroutine

The **openx** subroutine is intended primarily for use by diagnostic commands and utilities. Appropriate authority is required for execution. Attempting to execute this subroutine without the proper authority results in a return value of -1, with the **errno** global variable set to **EACCES**.

The *ext* parameter passed to the **openx** subroutine selects the operation to be used for the target adapter. The **/usr/include/sys/scsi.h** file defines the following possible values for the *ext* parameter:

Value	Description
SC_DIAGNOSTIC	Places the selected adapter in Diagnostic mode. This mode is singularly entrant. When an adapter is in Diagnostic mode, all controllers and DASD on that adapter are placed in Diagnostic mode, and error logging is disabled.

Value	Description
SD_DAEMON	Informs the device driver that the serial DASD subsystem daemon is opening the adapter. Only one Daemon-mode open is allowed per adapter, and it must be directed to the **/dev/dserdasda***n* special file.

This mode of operation is transparent to the rest of the system. A diagnostic open does not fail if the only open to an adapter is the daemon open, and the daemon open does not fail if an adapter is in Diagnostic mode. However, when an adapter is placed in Diagnostic mode, all queuing of asynchronous events to the daemon is filtered. The Daemon mode provides the authority needed to perform microcode downloads to the adapter and controller.

ioctl Subroutine

The **ioctl** subroutine can call the **IOCINFO**, **SD_SCSICMD**, and **SD_RESET** operations.

IOCINFO

The **IOCINFO** operation is the only operation defined for all device drivers that use the **ioctl** subroutine. The **IOCINFO** operation returns the **devinfo** structure defined in the **/usr/include/sys/devinfo.h** file. This ioctl operation can be directed to an adapter, controller, or DASD. The device can be opened in normal mode for this ioctl operation.

SD_SCSICMD

When the device has been successfully opened in Diagnostic mode, the **SD_SCSICMD** operation provides the means for issuing any Serial DASD subsystem command to a specified device. The Serial DASD subsystem commands are modeled after those for SCSI. The following Serial DASD subsystem commands are valid and use the same command descriptor block, including the operation code, as their corresponding SCSI command. The SCSI commands are defined in the **/usr/include/sys/scsi.h** file:

- Format Unit
- Inquiry
- Mode Select
- Mode Sense
- Read (6)
- Read (10)
- Read Capacity
- Reassign Blocks
- Release
- Request Sense
- Reserve
- Start Stop Unit
- Test Unit Ready
- Verify
- Write
- Write And Verify
- Write Buffer
- Write Sense

The following Serial DASD subsystem commands can be issued when the device is not in Diagnostic mode, but the caller has root authority:

- Fence
- Inquiry
- Read

- Read Extended
- Receive Diagnostics
- Request Sense
- Write Buffer

The Serial DASD Fence command does not correspond with any SCSI command. For more information, see ″Serial DASD Fence Command″ .

If the **SD_SCSICMD** operation is issued with any other Serial DASD subsystem command and the device is not in Diagnostic mode, the **SD_SCSICMD** operation returns a value of -1 and sets the **errno** global variable to a value of **EACCES**. The device driver performs no error recovery or error logging when this ioctl operation fails. The **status_validity** byte, **scsi_bus_status** byte, and the **adapter_status** byte are returned via the *arg* parameter. This parameter contains the address of a **sc_iocmd** structure, which is defined in the **/usr/include/sys/scsi.h** file.

The **devinfo** structure, which is returned from the **IOCINFO** ioctl operation, defines the maximum transfer size for the command. The structure returns a value of -1 and sets the **errno** global variable to a value of **EINVAL** if an attempt is made to transfer more than the maximum transfer size. If the Serial DASD subsystem command cannot complete in the time specified in the **sc_iocmd** structure, a -1 value is returned and the **errno** global variable is set to a value of **ETIMEDOUT**.

The **SD_SCSICMD** operation uses the **sc_iocmd** structure with the following status validity values:

Value	Description
0x00	Command successful
0x01	Valid **scsi_bus_status** byte only
0x02	Valid adapter status only
0x04	Valid alert register contents

Note: Except when the adapter status is **SC_ADAPTER_HDW_FAILURE**, if the adapter status is valid, the alert register contents are also valid. When the device driver fails a command due to internal error recovery without communicating with the adapter, the **SC_ADAPTER_HDW_FAILURE** adapter status is returned. For all other adapter status values, the adapter status is defined as part of the general card status in the **/usr/include/sys/scsi.h** file.

The **SD_SCSICMD** operation also uses the following reserved fields:

Field	Description
resvd1	Returned as the controller status byte of the alert register. The possible values for this field are defined in the ″Controller Status Byte Codes″ section of the **/usr/include/sys/serdasd.h** file.
resvd2	Returned as the adapter status byte of the alert register. The possible values for this field are defined in the ″Adapter Status Byte Codes″ section of the **/usr/include/sys/serdasd.h** file.
resvd5	Specifies the address of the device. The address is comprised of eight bits
5-7	Target (Controller ID)
4	Direction (0 = DASD, 1 = Controller)
	0-3 DASD (LUN)
resvd6	Manipulates the queue control and ordering of the **SD_SCSICMD** command. The caller must set this field to one of the following values:
0x00	None, unqueued
0x40	Invalid
0x80	Ordered
	0xC0 Unordered
resvd7	Specifies an extension to the Serial DASD subsystem command. The caller must set this field to one of the following values:

Field	Description
0x00	No extension
0x20	Split write enabled
	0x10 Split read enabled

Note: If the `timeout` field of the **sc_iocmd** structure is set to 0, the calling process is responsible for handling timeouts. The device driver will never time out the command if the value of the structure is 0.

SD_RESET

The **SD_RESET** operation provides an interface for issuing resets and quiesces to the Serial DASD adapter. The adapter is the **/dev/serdasda***N* file, where *N* is 0 or a positive integer. To send a reset or quiesce operation to an adapter, the adapter must be opened in **SC_DIAGNOSTIC** mode. The *arg* parameter of the **ioctl** call is a pointer to a **sd_ioctl_parms** structure, which is defined in the **/usr/include/sys/serdasd.h** file.

If the **SD_RESET** ioctl is issued when the device is not in Diagnostic mode, the ioctl returns a value of -1 and sets the **errno** global variable to a value of **EACCES**. If the Serial DASD adapter does not respond to the **SD_RESET** ioctl, a value of -1 is return, and the **errno** global variable is set to a value of **EIO**.

The following fields of the **sd_ioctl_parms** structure are used with this ioctl command.

Field	Description
`reset_type`	Contains one of the following values: **SD_RESET_OP** Indicates a full reset. **SD_QUIESCE_OP** Indicates a quiesce.
`status_validity`	Indicates either successful completion or the status of the adapter or controller. This field may have one of the following values: **0x00** Command successful **0x01** Valid adapter status **0x02** Valid controller status **0x04** Valid device driver status only **Note:** These values are different than the `status_validity` definitions for pass-through Serial DASD subsystem commands.The valid device driver status is exclusive of the other two valid status conditions. If the value of the status validity field is **0x04**, then the `adapter_status` field contains the device driver status. the only device driver status is **SD_DD_PURGED_TAG**
`adapter_status`	Indicates the adapter status, if valid, upon completion of the reset operation. The possible values for this field are defined in the ″Adapter Status Byte Codes″ section of the **/usr/include/sys/serdasd.h** file.
`controller_status`	Indicates the controller status, if valid, upon completion of the reset operation. The possible values for this field are defined in the ″Controller Status Byte Codes″ section of the **/usr/include/sys/serdasd.h** file.

Note: If the `time_out` field of the **sd_ioctl_parms** structure is set to 0, the calling process is responsible for handling timeouts. In this situation, the device driver will never time out the command.

Related Information

The **ioctl** subroutine, **open**, **openx** subroutine.

Serial DASD Subsystem Device Driver.

Device-Dependent Subroutines for Serial DASD Operations.

Device-Dependent Subroutines for Serial DASD Controller Operations.

Error Conditions for Serial DASD Subroutines.

Reliability, Availability, and Serviceability (RAS) Daemon for the Serial DASD Subsystem.

Serial DASD Fence Command.

Serial DASD Concurrent Mode of Operation Interface.

Serial Direct Access Storage Device (DASD) Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Error Conditions for Serial DASD Subroutines

In addition to those errors listed, **ioctl**, **open**, **read**, and **write** subroutines against this device are not successful in the following circumstances:

Value	Description
EACCES	Indicates that an attempt was made to open a device currently opened in Diagnostic mode.
EACCES	Indicates that an attempt was made to open a diagnostic session on a device already opened.
EACCES	Indicates that an attempt was made to open a device whose parent device is currently opened in Diagnostic mode.
EACCES	Indicates that an attempt was made to open a diagnostic session on a device whose child devices are already opened.
EACCES	Indicates that a diagnostic ioctl operation was attempted when the device was not in Diagnostic mode.
EACCES	Indicates that a daemon ioctl operation was attempted by a process other than the RAS configuration daemon, or the adapter has not been opened by the daemon.
EBUSY	Indicates that the target device is reserved by another initiator.
EBUSY	Indicates that the other initiator may have outstanding requests queued to the device.
EBUSY	Indicates that a device cannot be unconfigured if it is still in use.
EFAULT	Indicates that a severe I/O error occurred during an adapter download.
EINVAL	Indicates that the **read** or **write** subroutine supplied an *nbyte* parameter that is not an even multiple of the block size.
EINVAL	Indicates that an unsupported ioctl operation was attempted.
EMEDIA	Indicates that the target device has indicated an unrecovered media error.
ESOFT	Indicates that the target device has indicated a recovered media error.
ENXIO	Indicates that the **ioctl** subroutine supplied an invalid parameter.
ENXIO	Indicates that a **read** or **write** command was attempted beyond the end of the disk.
EIO	Indicates that the target device cannot be located or is not responding.
EIO	Indicates that the target device has indicated an unrecovered hardware error.
EPERM	Indicates that the attempted subroutine requires appropriate authority.
ETIMEDOUT	Indicates that an ioctl operation timed out.
ENOMEM	Indicates that there is insufficient memory to perform the request.
ECHILD	Indicates that there are no more asynchronous events needing processing by the RAS configuration daemon.

Related Information

Serial DASD Subsystem Device Driver.

Device-dependent Subroutines for Serial DASD Operations.

Device-Dependent Subroutines for Serial DASD Controller Operations.

Device-Dependent Subroutines for Serial DASD Adapter Operations.

Reliability, Availability, and Serviceability (RAS) Daemon for the Serial DASD Subsystem.

The **ioctl** subroutine, **open** subroutine, **read** subroutine, **write** subroutine.

Reliability, Availability, and Serviceability (RAS) Daemon for the Serial DASD Subsystem

The RAS (Reliability, Availability, and Serviceability) daemon for the serial Direct Access Storage Device (DASD) subsystem provides off-level error handling support to the device driver. The main function of the daemon is to recognize signals from the device driver and take the appropriate action. One such action is to verify the downloadable microcode levels within the subsystem and to download appropriate levels again if necessary. This way, even errors within the subsystem causing the loss of downloaded microcode can be recovered.

The RAS daemon appears in the process table as **sdd** (serial DASD daemon). There is one daemon running for each serial DASD adapter in the system. This process should not be killed because it is there for enhanced data integrity and error recovery capabilities. The daemon cannot be respawned without unconfiguring and reconfiguring the affected adapter.

Related Information

The **/dev/rhdisk** special file, **/dev/serdasda** special file, **/dev/serdasdc** special file.

The **close** subroutine, **ioctl** subroutine, the **lseek** subroutine, **open** or **openx** subroutine, **read**, **readx** subroutine, **write**, **writex** subroutine.

Serial DASD Fence Command

The Serial Direct Acess Storage Device (DASD) controller supports up to eight hosts and can *fence*, or lock out, specified hosts. Fences are established and removed via the Serial DASD subsystem Fence command. Once a fence is established, it can only be removed with another Fence command or by cycling the power to the controller.

Hardware Implementation

Each DASD has an associated two-byte fence register in the controller. A bit set to 1 indicates the host attached to the specified host connector on the back of the controller drawer can only issue the Inquiry, Request Sense, Fence, and Read(10) Serial DASD subsystem commands. The read with reservation (RWR) bit must be set in the command descriptor block. The host connectors on the back of the controller drawer are labeled from 0-7, and the fence register bits are ordered from left to right. The host connector bit is 0.

The following table illustrates the Serial DASD Fence command descriptor block:

Byte	Bit							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	Operation Code = D0h							
1	LUN = 0			Force Fence	Command Modifier			
2 and 3	Fence Mask							
4 and 5	Fence Data							
6 and 7	Reserved = 0							
8	Allocation Length = 4							
9	VU = 0		Reserved = 0				Flag	Link

The command descriptor block for the Fence command contains the following fields:

Field — **Description**

`Operation Code` — Specifies the operation code for this command. The value of this field is always D0h.

`LUN` — Logical unit number. The value of this field is always 000.

`Force Fence` — Specifies how to run the Fence command. It can have the following values:

0 Runs the Fence command from a host that is not fenced-out.

1 Runs the Fence command from a fenced-out host.

`Command Modifier` — Specifies which type of fencing operations is to be used. The command modifier can have one of the following values:

0001 Indicates the Mask and Swap operation performs the following equation in order to set the DASD's fence bit register:

(fence data field & fence mask field) | (old fence register value & ~fence mask field)

That is, the fence mask field specifies which bits in the fence data field should be used.

The Mask and Swap operation allows fences for multiple hosts to complement one another, since they do not need to know the current state of the fence to fence out another host. For example, if host0 and host3 are currently fenced out, then the fence register will have the value of 1001000000000000. To unfence host3 using Mask and Swap requires the fence data field to be set to 0000000000000000 and the fence mask field to be set to 0001000000000000.

0010 Indicates the Compare and Swap fencing operation sets the DASD's fence bit register, depending on the value of the fence mask field. If the old fence bit register equals fence mask field, the fence bit register is set equal to the fence data field. Otherwise, the fence bit register remains unchanged.

`Fence mask` — Indicates a 16-bit fence mask field. Its value is dependent on the value of the Command Modifier component.

`Fence data` — Indicates a 16-bit fence data field. Its value is dependent on the value of the Command Modifier component.

`Reserved` — The value of this field is always 0000000000000000.

`Allocation Length` — The value of this field is always 00000100.

`VU` — The value of this field is always 00.

`Reserved` — The value of this field is always 0000.

`Flag` — The value of this field is always 0.

`Link` — The value of this field is always 0.

The Fence command also supplies a method to determine which hosts are currently fenced out as well as which tail the current host is connected via the data returned from the Fence command. The following on table illustrates the data returned by the Fence command.

	Bit							
Byte	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0 and 1	Fence Position Indicator							
2 and 3	(Old) Fence Value							

A fence cannot be removed by a reset. It can only be removed by cycling power on the controller of the DASD, or by issuing a Fence command.

Software Implementation

The Serial DASD subsystem device driver uses the hardware's Mask and Swap fence commands to set and remove fences. The device driver also uses the Mask and Swap fence command with the fence mask set to all zeroes to determine its current host position.

The Serial DASD controller configuration method enables the fencing mechanism. Once enabled, a user can create or remove fences with ioctls to individual DASDs. The device driver maintains a fence by reestablishing it whenever the DASD is powered off and on.

Related Information

Serial DASD Subsystem Device Driver.

Serial DASD Concurrent Mode of Operation Interface.

Device-Dependent Subroutines for Serial DASD Operations.

Device-Dependent Subroutines for Serial DASD Adapter Operations.

Device-Dependent Subroutines for Serial DASD Controller Operations.

Serial Direct Access Storage Device (DASD) Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Serial DASD Concurrent Mode of Operation Interface

The Serial Direct Acess Storage Device (DASD) subsystem supports the ability to broadcast one-byte message codes from one host to all other hosts connected to the same DASD. This message-passing capability can be used to synchronize access to the DASD. The operating system has a concurrent mode interface to use this hardware functionality.

The concurrent mode of operation requires that a top kernel extension runs on all hosts sharing a DASD. The top kernel extensions communicate with each other via the Serial DASD subsystem using the concurrent mode interface of the Serial DASD subsystem device driver. This interface allows a top kernel extension to send and receive messages between hosts.

The concurrent mode interface consists of an entry point in both the Serial DASD device driver and the top kernel extension. Two ioctls register and unregister the top kernel extension with the Serial DASD device driver. The Serial DASD device driver's entry point provides the means to send messages as well as lock, unlock, and test DASD. The top kernel extension entry point processes interrupts, including receiving messages from other hosts.

Device Driver Entry Point

The Serial DASD device driver concurrent mode entry point sends commands from the top kernel extension for a specified Serial DASD. The top kernel extension calls this entry point directly. The **DD_CONC_REGISTER** ioctl operation registers entry points.

This entry point function takes one argument, which is a pointer to a **conc_cmd** structure, which is defined in the **/usr/include/sys/ddconc.h** file. The **conc_cmd** structures must be allocated by the top kernel extension. The concurrent mode command operation is specified by the `cmd_op` field in the **conc_cmd** structure and can have the following values. For each operation, the `devno` field of the **conc_cmd** structure specifies the appropriate Serial DASD.

Value	Description
DD_CONC_SEND_REFRESH	Broadcasts the one-byte message code specified by the `message` field of the **conc_cmd** structure. The code is sent to all hosts connected to the Serial DASD.
DD_CONC_LOCK	Locks the specified Serial DASD for this host only. No other hosts will be able to modify data on the DASD.
DD_CONC_UNLOCK	Unlocks the Serial DASD. Other hosts can lock and modify data on the DASD.
DD_CONC_TEST	Issues a test DASD command to verify that the Serial DASD is still accessible to this host.

The concurrent mode entry point returns a value of **EINVAL** if any of the following are true:

- The top kernel extension did not perform a **DD_CONC_REGISTER** operation.
- The **conc_cmd** pointer is null.
- The `devno` field in the **conc_cmd** structure is invalid.
- The `cmd_op` field in the **conc_cmd** structure is not one of the four valid values previously listed.

If the concurrent mode entry point accepts the **conc_cmd** structure, the entry point returns a value of 0. If the Serial DASD device driver does not have resources to issue the command, the driver queues the command until resources are available. The concurrent commands queued in the Serial DASD device driver are issued before any read or write operations queued by the driver's strategy entry point.

The completion status of the concurrent mode commands are returned to the top kernel extension's concurrent mode interrupt handler entry point.

Top Kernel Extension Entry Point

The top kernel extension must have a concurrent mode command interrupt handler entry point, which is called directly from the Serial DASD subsystem device driver's interrupt handler. This function can take four arguments: the **conc_cmd** pointer, and the `cmd_op`, `message_code`, and `devno` fields. The **conc_cmd** pointer points to a **conc_cmd** structure. These arguments must be of the same type specified by the `conc_intr_addr` function pointer field in the **dd_conc_register** structure.

The following valid concurrent mode commands are defined in the **/usr/include/sys/ddcon.h** file. For each, the `devno` field specifies the appropriate Serial DASD.

Command	Description
DD_CONC_SEND_REFRESH	Indicates the **DD_CONC_SEND_REFRESH** device driver entry point completed. The error field in the **conc_cmd** structure contains the return code necessary for the completion of this command. The possible values are defined in the **/usr/include/sys/errno.h** file. The **conc_cmd** pointer argument to the top kernel extension's special interrupt handler entry point is non-null. The `cmd_op`, `message_code`, and `devno` fields are 0.

Command	Description
DD_CONC_LOCK	Indicates the **DD_CONC_SEND_LOCK** device driver entry point completed. The error field in the **conc_cmd** structure contains the return code necessary for the completion of this command. The possible values are defined in the **/usr/include/sys/errno.h** file. The **conc_cmd** pointer argument to the top kernel extension's special interrupt handler entry point is non-null. The `cmd_op`, `message_code`, and `devno` fields are zero.
DD_CONC_UNLOCK	Indicates the **DD_CONC_UNLOCK** device driver entry point completed. The error field in the **conc_cmd** structure contains the return code necessary for the completion of this command. The possible values are defined in the **/usr/include/sys/errno.h** file. The **conc_cmd** pointer argument to the top kernel extension's special interrupt handler entry point is non-null. The `cmd_op`, `message_code`, and `devno` fields are zero.
DD_CONC_TEST	Indicates the **DD_CONC_TEST** device driver entry point completed. The error field in the **conc_cmd** structure contains the return code necessary for the completion of this command. The possible values are defined in the **/usr/include/sys/errno.h** file. The **conc_cmd** pointer argument to the top kernel extension's special interrupt handler entry point is non-null. The `cmd_op`, `message_code`, and `devno` fields are zero.
DD_CONC_RECV_REFRESH	Indicates a message with **message_code** was received for the DASD specified by the **devno** argument. The **conc_cmd** argument is null for this operation.
DD_CONC_RESET	Indicates the DASD specified by the **devno** argument was reset, and all pending messages or commands have been flushed. The argument **conc_cmd** is null for this operation.

A kernel extension (referred to as the top kernel extension), using the concurrent mode of operation with the Serial DASD subsystem device driver, must meet the following requirements:

- The concurrent command interrupt handler routine must have a short path length since it will be running on the Serial DASD subsystem interrupt level. If substantial command processing is needed, then this routine should schedule an off-level interrupt to its own off-level interrupt handler.
- The concurrent command interrupt handler routine must not call the Serial DASD device driver's concurrent entry point directly. If it is necessary to call the Serial DASD device driver's concurrent entry point, than this routine (the concurrent command interrupt handler routine) should schedule an off-level interrupt to it's own off-level interrupt handler to make the call.
- The top kernel extension must have an interrupt priority no higher than the DASD device driver's interrupt priority.
- The concurrent command interrupt handler routine may need to disable interrupts at INTCLASS0 if it is expected to use concurrent mode on Serial DASD and some DASD of different types. The other type of DASD needs its own device driver to support the concurrent mode.
- A kernel extension that uses the **DD_CONC_REGISTER** ioctl must issue a **DD_CONC_UNREGISTER** ioctl before closing the device.

Related Information

Serial Direct Access Storage Device (DASD) Overview in *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

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 give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: 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 states 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.

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 Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.

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.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

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 the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates 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. 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 IBM's application programming interfaces.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
IBM
Micro Channel
PowerPC

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.

Index

Special characters

A

B

C

D

E

F

G

I

K

L

M

O

P

Q

R

S

T

U

V

W

Vos remarques sur ce document / Technical publication remark form

Titre / Title : Bull AIX 5L Technical Reference Kernel and Subsystems Volume 2/2

Nº Reférence / Reference Nº : 86 A2 52EF 02

Daté / Dated : May 2003

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement.
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : _____ Date : _____

SOCIETE / COMPANY : _____

ADRESSE / ADDRESS : _____

_____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

Technical Publications Ordering Form

Bon de Commande de Documents Techniques

To order additional publications, please fill up a copy of this form and send it via mail to:
Pour commander des documents techniques, remplissez une copie de ce formulaire et envoyez-la à :

BULL CEDOC
ATTN / Mr. L. CHERUBIN
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

Phone / Téléphone : +33 (0) 2 41 73 63 96
FAX / Télécopie +33 (0) 2 41 73 60 19
E–Mail / Courrier Electronique : srv.Cedoc@franp.bull.fr

Or visit our web sites at: / Ou visitez nos sites web à:
http://www.logistics.bull.net/cedoc
`http://www-frec.bull.com` `http://www.bull.com`

CEDOC Reference # Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]		_ _ _ _ _ _ _ _ _ [_ _]	
[_ _] : **no revision number means latest revision** / pas de numéro de révision signifie révision la plus récente					

NOM / NAME : ____________________ Date : __________

SOCIETE / COMPANY : ____________________

ADRESSE / ADDRESS : ____________________

____

PHONE / TELEPHONE : __________ FAX : __________

E–MAIL : ____________________

For Bull Subsidiaries / Pour les Filiales Bull :
Identification: ____________________

For Bull Affiliated Customers / Pour les Clients Affiliés Bull :
Customer Code / Code Client : ____________________

For Bull Internal Customers / Pour les Clients Internes Bull :
Budgetary Section / Section Budgétaire : ____________________

For Others / Pour les Autres :
Please ask your Bull representative. / Merci de demander à votre contact Bull.

PLACE BAR CODE IN LOWER LEFT CORNER

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 52EF 02

Bull

Utiliser les marques de découpe pour obtenir les étiquettes.
Use the cut marks to get the labels.

AIX
AIX 5L Technical Reference Kernel and Subsystems
Volume 2/2
86 A2 52EF 02

AIX
AIX 5L Technical Reference Kernel and Subsystems
Volume 2/2
86 A2 52EF 02

AIX
AIX 5L Technical Reference Kernel and Subsystems
Volume 2/2
86 A2 52EF 02