AIX Version 7.2

Technical Reference: Kernel and Subsystems, Volume 2

IBM

AIX Version 7.2

Technical Reference: Kernel and Subsystems, Volume 2

Note

Before using this information and the product it supports, read the information in “Notices” on page 289.

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

Contents

About this document vii
Highlighting vii
Case sensitivity in AIX vii
ISO 9000 vii

Technical Reference: Kernel and Subsystems, Volume 2 1
What's new in Technical Reference: Kernel and Subsystems, Volume 2 1
Configuration Subsystem 1
Adapter-Specific Considerations for the Predefined Attribute (PdAt) Object Class 1
Adapter-Specific Considerations for the Predefined Devices (PdDv) Object Class 4
attrval Device Configuration Subroutine 5
busresolve Device Configuration Subroutine . . . 6
Configuration Rules (Config_Rules) Object Class . 8
Customized Attribute (CuAt) Object Class . . . 9
Customized Dependency (CuDep) Object Class 11
Customized Device Driver (CuDvDr) Object Class 11
Customized Devices (CuDv) Object Class . . . 12
Customized VPD (CuVPD) Object Class 15
Device Methods for Adapter Cards: Guidelines 16
genmajor Device Configuration Subroutine . . . 16
genminor Device Configuration Subroutine. . . 17
genseq Device Configuration Subroutine. . . . 19
getattr Device Configuration Subroutine 20
getminor Device Configuration Subroutine . . . 21
How Device Methods Return Errors 22
loadext Device Configuration Subroutine . . . 23
Loading a Device Driver 24
Machine Device Driver 24
ODM Device Configuration Object Classes . . . 31
Predefined Attribute (PdAt) Object Class . . . 32
Predefined Attribute Extended (PdAtXtd) Object Class 36
Predefined Connection (PdCn) Object Class . . 38
Predefined Devices (PdDv) Object Class 38
putattr Device Configuration Subroutine. . . . 42
reldevno Device Configuration Subroutine . . . 43
relmajor Device Configuration Subroutine . . . 44
Writing a Change Method 45
Writing a Configure Method. 47
Writing a Define Method 51
Writing an Unconfigure Method 54
Writing an Undefine Method 56
Writing Optional Start and Stop Methods . . . 57
Communications Subsystem 58
CIO_GET_FASTWRT ddioctl Communications PDH Operation 58
CIO_GET_STAT (Get Status) tsioctl PCI MPQP Device Handler Operation 59
CIO_GET_STAT ddioctl Communications PDH Operation 62
CIO_HALT (Halt Device) tsioctl PCI MPQP Device Handler Operation 64
CIO_HALT ddioctl Communications PDH Operation 65
CIO_QUERY (Query Statistics) tsioctl PCI MPQP Device Handler Operation 66
CIO_QUERY ddioctl Communications PDH Operation 68
CIO_START (Start Device) tsioctl PCI MPQP Device Handler Operation 69
CIO_START ddioctl Communications PDH Operation 73
dd_fastwrt Communications PDH Entry Point. . 74
ddclose Communications PDH Entry Point . . . 75
ddopen (Kernel Mode) Communications PDH Entry Point 76
ddopen (User Mode) Communications PDH Entry Point 78
ddread Communications PDH Entry Point . . . 79
ddselect Communications PDH Entry Point . . 81
ddwrite Communications PDH Entry Point. . . 82
MP_CHG_PARMS (Change Parameters) tsioctl PCI MPQP Device Handler Operation 84
tsclose Multiprotocol (PCI MPQP) Device Handler Entry Point 84
tsconfig Multiprotocol (PCI MPQP) Device Handler Entry Point 86
tsioctl Multiprotocol (PCI MPQP) Device Handler Entry Point 86
tsmpx Multiprotocol (PCI MPQP) Device Handler Entry Point 88
tsopen Multiprotocol (PCI MPQP) Device Handler Entry Point 89
tsread Multiprotocol (PCI MPQP) Device Handler Entry Point 90
tsselect Multiprotocol (PCI MPQP) Device Handler Entry Point 92
tswrite Multiprotocol (PCI MPQP) Device Handler Entry Point 93
LFT Subsystem 95
dd_close LFT Device Driver Interface. 95
dd_ioctl LFT Device Driver Interface 96
dd_open LFT Device Driver Interface. 96
DIALREGRING (Register Input Ring) 97
DIALRFLUSH (Flush Input Ring) 97
DIALSETGRAND (Set Dial Granularity). . . . 98
GIOQUERYID (Query Attached Devices) . . . 98
Input Device Driver ioctl Operations 99
IOCINFO (Return devinfo Structure) ioctl Input Device Driver 100
KSALARM (Sound Alarm) 100
KSCFGCLICK (Enable/Disable Keyboard Clicker) 101
KSDIAGMODE (Enable/Disable Diagnostics Mode). 102
KSKAP (Enable/Disable Keep Alive Poll) . . . 102

KSKAPACK (Acknowledge Keep Alive Poll) 103
KSLED (Illuminate/Darken Keyboard LEDs) 104
KSQUERYID (Query Keyboard Device Identifier) 104
KSQUERYSV (Query Keyboard Service Vector) 105
KSREGRING (Register Input Ring) 106
KSRFLUSH (Flush Input Ring) 107
KSTDELAY (Set Typematic Delay) 107
KSTRATE (Set Typematic Rate) 108
KSVOLUME (Set Alarm Volume) ioctl 108
lft_dds_t Structure. 109
lft_t Structure 111
LPFKLIGHT (Set/Reset Key Lights) 112
LPFKREGRING (Register Input Ring) 112
LPFKRFLUSH (Flush Input Ring). 113
MQUERYID (Query Mouse Device Identifier) 113
MREGRING (Register Input Ring) 114
MREGRINGEXT (Register Extended Input Ring) 114
MRESOLUTION (Set Mouse Resolution) . . . 115
MRFLUSH (Flush Input Ring) 115
MSAMPLERATE (Set Mouse Sample Rate) . . 116
MSCALE (Set Mouse Scale Factor) 116
MTHRESHOLD (Set Mouse Reporting Threshold) 117
phys_displays Structure 117
TABCONVERSION (Set Tablet Conversion Mode) 121
TABDEADZONE (Set Tablet Dead Zone) . . . 121
TABORIGIN (Set Tablet Origin) 122
TABQUERYID (Query Tablet Device Identifier) ioctl Tablet Device Driver Operation. 122
TABREGRING (Register Input Ring). 123
TABRESOLUTION (Set Tablet Resolution) . . . 123
TABRFLUSH (Flush Input Ring) 124
TABSAMPLERATE (Set Tablet Sample Rate) ioctl Tablet Device Driver Operation. 124
Virtual Display Driver (VDD) Interface (lftvi) 125
vtmstruct Structure 127
Printer Subsystems 128
passthru Subroutine 128
piocmdout Subroutine 129
pioexit Subroutine. 130
piogetattrs Subroutine 131
piogetopt Subroutine 132
piogetstatus Subroutine 133
piogetstr Subroutine 134
piogetvals Subroutine 135
piomsgout Subroutine 137
pioputattrs Subroutine 138
pioputstatus Subroutine 139
restore Subroutine. 140
setup Subroutine 141
Subroutines for Print Formatters 142
Subroutines for Writing a Print Formatter . . . 142
SCSI Subsystem 142
IOCINFO (Device Information) tmscsi Device Driver ioctl Operation 142
Parallel SCSI Adapter Device Driver. 143
scdisk SCSI Device Driver 151
SCIOCMD SCSI Adapter Device Driver ioctl Operation 169
SCIODIAG (Diagnostic) SCSI Adapter Device Driver ioctl Operation 171
SCIODNLD (Download) SCSI Adapter Device Driver ioctl Operation 172
SCIOEVENT (Event) SCSI Adapter Device Driver ioctl Operation 173
SCIOGTHW (Gathered Write) SCSI Adapter Device Driver ioctl Operation 175
SCIOHALT (Halt) SCSI Adapter Device Driver ioctl Operation 175
SCIOINQU (Inquiry) SCSI Adapter Device Driver ioctl Operation 176
SCIOREAD (Read) SCSI Adapter Device Driver ioctl Operation 177
SCIORESET (Reset) SCSI Adapter Device Driver ioctl Operation 179
SCIOSTART (Start SCSI) Adapter Device Driver ioctl Operation 180
SCIOSTARTTGT (Start Target) SCSI Adapter Device Driver ioctl Operation 181
SCIOSTOP (Stop) Device SCSI Adapter Device Driver ioctl Operation 183
SCIOSTOPTGT (Stop Target) SCSI Adapter Device Driver ioctl Operation 184
SCIOSTUNIT (Start Unit) SCSI Adapter Device Driver ioctl Operation 185
SCIOTRAM (Diagnostic) SCSI Adapter Device Driver ioctl Operation 186
SCIOTUR (Test Unit Ready) SCSI Adapter Device Driver ioctl Operation 187
scsesdd SCSI Device Driver 188
scsidisk SAM Device Driver 190
scsisesdd SAM Device Driver 206
sctape FC Device Driver. 209
tape SCSI Device Driver. 215
TMCHGIMPARM (Change Parameters) tmscsi Device Driver ioctl Operation 221
TMGETSENS (Request Sense) tmscsi Device Driver ioctl Operation 222
TMIOASYNC (Async) tmscsi Device Driver ioctl Operation 223
TMIOCMD (Direct) tmscsi Device Driver ioctl Operation 223
TMIOEVNT (Event) tmscsi Device Driver ioctl Operation 224
TMIORESET (Reset Device) tmscsi Device Driver ioctl Operation 226
TMIOSTAT (Status) tmscsi Device Driver ioctl Operation 226
tmscsi SCSI Device Driver 227
| NVMe subsystem 233
| NVMe storage (hdisk) device driver. 234
| NVMe controller device driver 235
USB Subsystem. 236
Extensible Host Controller Adapter Device Driver 236
Enhanced Host Controller Adapter Device Driver 240
Open Host Controller Adapter Device Driver 242
HCD_REGISTER_HC Adapter Device Driver ioctl Operation 245

USB Audio Device Driver 245
USB Keyboard Client Device Driver 246
USB Mass Storage Client Device Driver 248
USB Mouse Client Device Driver 264
USB Tape Client Device Driver 265
USBD Protocol Driver 276
USBLIBDD Passthru Driver 284

Notices 289
Privacy policy considerations 291
Trademarks 291

Index 293

About this document

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

Highlighting

The following highlighting conventions are used in this document:

Bold
Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Bold highlighting also identifies graphical objects, such as buttons, labels, and icons that the you select.

Italics
Identifies parameters for actual names or values that you supply.

`Monospace`
Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or text that you must type.

Case sensitivity in AIX

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

ISO 9000

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

Technical Reference: Kernel and Subsystems, Volume 2

This topic collection provides information about subroutines in the Configuration Subsystem, Communications Subsystem, Low function terminal (LFT) Subsystem, Printer Subsystems, Integrated Device Electronics (IDE), and other Subsystems.

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

What's new in Technical Reference: Kernel and Subsystems, Volume 2

Read about new or significantly changed information for the Technical Reference: Kernel and Subsystems, Volume 2 topic collection.

How to see what's new or changed

In this PDF file, you might see revision bars (|) in the left margin, which identify new and changed information.

March 2018

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

- Added information about "NVMe subsystem" on page 233 topic.
- Added information about "NVMe storage (hdisk) device driver" on page 234 topic.
- Added information about "NVMe controller device driver" on page 235 topic.

Configuration Subsystem

System users cannot operate devices until device configuration occurs. To configure devices, the Device Configuration Subsystem is available.

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 reference:

"Predefined Attribute (PdAt) Object Class" on page 32

"Writing a Change Method" on page 45

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

Related information:

Understanding Interrupts

Object Data Manager (ODM) Overview for Programmers

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:

Item	Description
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 reference:

"Adapter-Specific Considerations for the Predefined Attribute (PdAt) Object Class" on page 1
"Writing a Configure Method" on page 47
"Predefined Devices (PdDv) Object Class" on page 38

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

Item	Description
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

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

Files

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

Related reference:

"Predefined Attribute (PdAt) Object Class" on page 32

"Customized Attribute (CuAt) Object Class" on page 9

Related information:

List of Device Configuration Subroutines

busresolve Device Configuration Subroutine

Purpose

Allocates bus resources for adapters on an I/O bus.

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

Item	Description
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

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 must 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 it 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 also contains a list of the logical names of the devices for which it can not successfully resolve all attributes. Devices whose names appear in the *not_res_list* parameter must not be configured into the Available state.

When you write a Configure method for a device that has 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 might be insufficient space to hold the device names.

Return Values

Item	Description
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 database.
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

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

Related reference:

"ODM Device Configuration Object Classes" on page 31

Related information:

Understanding ODM Object Classes and Objects

List of Device Configuration Subroutines

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 *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.

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 reference:

"Writing a Configure Method" on page 47

Related information:

bosboot command

Device Configuration Manager Overview

Object Data Management (ODM) Overview for Programmers

System boot processing

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 reference:

"ODM Device Configuration Object Classes" on page 31

"Customized Devices (CuDv) Object Class" on page 12

"Predefined Attribute (PdAt) Object Class" on page 32

"getattr Device Configuration Subroutine" on page 20

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.

Item	Description
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 reference:

"ODM Device Configuration Object Classes" on page 31

"Customized Devices (CuDv) Object Class" on page 12

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_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 reference:

"genmajor Device Configuration Subroutine" on page 16

"ODM Device Configuration Object Classes" on page 31

"Predefined Devices (PdDv) Object Class" on page 38

"getminor Device Configuration Subroutine" on page 21

"reldevno Device Configuration Subroutine" on page 43

Related information:

List of Device Configuration Subroutines

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.

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.

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.

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.

Descriptor | **Description**

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 reference:

“Predefined Devices (PdDv) Object Class” on page 38

“Writing a Define Method” on page 51

“Writing a Configure Method” on page 47

“Parallel SCSI Adapter Device Driver” on page 143

Related information:

Device Configuration Manager Overview

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 reference:

“ODM Device Configuration Object Classes” on page 31

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 *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 reference:

"ODM Device Configuration Object Classes" on page 31

Related information:

rmdev subroutine

Understanding Direct Memory Access (DMA)

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>

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

Parameters

Item	**Description**
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

Item	**Description**
/usr/lib/libcfg.a	Archive of device configuration subroutines.

Related reference:

"relmajor Device Configuration Subroutine" on page 44

"Customized Device Driver (CuDvDr) Object Class" on page 11

Related information:

List of ODM Commands and Subroutines

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>

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

Parameters

Item	Description
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

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

Related reference:

"getminor Device Configuration Subroutine" on page 21

"Customized Device Driver (CuDvDr) Object Class" on page 11

Related information:

List of ODM Commands and Subroutines

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

Item	Description
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

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

Related reference:

"Customized Devices (CuDv) Object Class" on page 12

Related information:

List of ODM Commands and Subroutines

List of Device Configuration Subroutines

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

Item	Description
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

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

Related reference:

"putattr Device Configuration Subroutine" on page 42

"Predefined Attribute (PdAt) Object Class" on page 32

Related information:

Device Configuration Subsystem Programming Introduction

Understanding ODM Object Classes and Objects

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>

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

Parameters

Item	Description
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

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

Related reference:

"genminor Device Configuration Subroutine" on page 17

"genmajor Device Configuration Subroutine" on page 16

"Customized Device Driver (CuDvDr) Object Class" on page 11

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 reference:

"Customized Attribute (CuAt) Object Class" on page 9

Related information:

Device Dependent Structure (DDS) Overview

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

Item	Description
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

Item	Description
/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:

sysconfig subroutine

List of Device Configuration Subroutines

Understanding Kernel Extension Binding

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

Item	Description
/usr/lib/drivers directory	Contains device drivers.

Related reference:

“loadext Device Configuration Subroutine” on page 23

“Predefined Devices (PdDv) Object Class” on page 38

“Predefined Attribute (PdAt) Object Class” on page 32

“Writing a Configure Method” on page 47

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 must not be used by general applications.

A program must have the appropriate privilege to open special files **/dev/nvram** or **/dev/bus***N*. 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 that is 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 that contains the base customize information is freed. Subsequent opens return an **ENOENT** error code.

read and write Subroutines

The **write** subroutine is not supported and returns 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 that is associated with the specified channel. The transfer starts at the offset (within the channel's data area) specified by the `offset` field that is 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 must point to a **mach_dd_io** structure. The `md_data` field must point to an integer; this field 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 must point to a **mach_dd_io** structure. The `md_data` field must point to an integer; this field 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.
MIONVGET	Reads data from an NVRAM address and returns data in the buffer that is provided by the caller. This operation is useful for reading the ROS area of NVRAM. A structure that defines 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 that is 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 cause the operation to fail with an **EINVAL** error code.

ioctl Operations for Systems

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

These **ioctl** operations can be called from user space or kernel space (by 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 is obtained to serialize access to the bus slot configuration register.

MIOVPDGET

This **ioctl** operation 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 or 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 is added to the base address to obtain the starting address for reading.

The **buc_info** structure for the selected bus slot is used to obtain the word increment value. This value performs correct addressing when it reads the data.

MIOCFGGET

This **ioctl** operation 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**, or **MV_WORD**).

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

MIOCFGPUT

This **ioctl** operation 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**, or **MV_WORD**).

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

MIORESET

This **ioctl** operation 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 the Common Hardware Reference Platform.

MIOGEARD

Scans for the variable name in the Global Environment Area, and, if found, the null terminated string is 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 is 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.
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	It is a pointer to the length of the returned global variable strings including the null terminators if md_length is non-zero.

MIOGEAUPD

The specified global variable is 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 replace the old after adjusting any size deltas. Further, any information that is moved toward a lower address has the original area zeroed. If there is no string that follows the variable name and equal sign, the specified variable is deleted. If the variable to be deleted is not found, a successful return occurs. The new information is written to **NVRAM**. Further, the header in the **NVRAM** operation is updated to include the update time of the Global Environment Area and the CRC value are 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	It is a pointer to the amount of space that is left in the Global Environment Area after the update. It is computed as the size of the area minus the length of all global variable strings minus the threshold value.

MIOGEAST

The specified threshold is set so that any updates done do not exceed the Global Environment Area size minus the threshold. In place of 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 are returned to the data area specified by the caller. The **gea_attrib** structure is 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 is 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.
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 is to 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 might be returned when you access 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 that is 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 that is 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 can 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 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 diagnostic tests are unable to access a device through the diagnostic functions that are provided by the device's own device driver, they might 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 that is 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 that is 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 that is 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

Item	Description
EFAULT	A buffer that is 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 can be allocated by the machine device driver for use in the data transfer.

Files

Item	Description
/dev/pci*N*	Provides access to the I/O bus (CHRP and the AIX operating system).
/dev/isa*N*	Provides access to the I/O bus (CHRP and the AIX operating system).
/dev/nvram	Provides access to platform-specific nonvolatile RAM.
/dev/nvram/base	Allows read access to the base customize information that is stored as part of the boot record.

Related information:

open subroutine

read subroutine

savebase command

bus file

ODM Device Configuration Object Classes

A list of the ODM Device Configuration Object Classes follows:

Item	Description
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 reference:

“busresolve Device Configuration Subroutine” on page 6

Related information:

Device Configuration Subsystem Programming Introduction

Writing a Device Method

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.

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.

Descriptor **Description**

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.

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.

Descriptor **Description**

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.

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`.

d Indicates that the default value for the attribute has been altered by the **chdef** command.

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 reference:

"ODM Device Configuration Object Classes" on page 31

"Predefined Devices (PdDv) Object Class" on page 38

"Customized Attribute (CuAt) Object Class" on page 9

Related information:

mkdev subroutine

Predefined Attribute Extended (PdAtXtd) Object Class

The Predefined Attribute Extended (PdAtXtd) object class is used to supplement existing device attributes that are represented in the Predefined Attribute (PdAt) object class with information that can be used by Device Management User Interface.

Types of attributes to represent in PdAtXtd

Not all existing device attributes in PdAt must be represented in the PdAtXtd object class. Non-displayable attributes (that is, attributes with a null string in the 'generic' field of the PdAt object class) must not have a corresponding PdAtXtd entry, otherwise, it becomes 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**	Attribute Classification	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**

uniquetype
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
Identifies the device attribute. This name can be passed to **mkdev** and **chdev** configuration commands and device methods in the attribute-name and attribute-value pairs.

classification
Identifies the classification of the device attribute. 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 that is used to position the attribute in relation to others on a panel or menu. This field is identical to the `id_seq_num` currently in the `sm_cmd_opt` (SMIT Dialog/Selector Command Option) object class.

operation
Identifies the type of operation that is associated with the unique device type. Operation and attribute name fields are mutually exclusive.

operation_value
Identifies the value that is associated with the `Operation` field.

When the operation is `add_`*`device`*, the `operation_value` field can contain the command that is used to make the device, if the **mkdev** command cannot be used.

description
Identifies the attribute description.

list_cmd
Identifies the command to override the current value of the attribute, except when the operation field is set. If the operation field is set, it identifies the command to return information that is associated with the operation.

For example, in the case of the `add_tty`operation, the `list_cmd` field contains the following value:

```
lsdev -P -c tty -s rs232 -Fdescription
```

list_values_cmd
Identifies the command to obtain the possible values of an attribute. The values that are returned override the values field in the Predefined Attribute object class.

change_cmd
Identifies the command to change the attribute value if the **chdev**command cannot be used.

help
Identifies the help text that is associated with the attribute. The help text format follows:

```
message file,set id,msg id,default text
```

OR

```
a numeric string equal to a SMIT identifier tag.
```

nls_values
Identifies the text that is associated with the attribute values. These values are displayed in place of the values that are stored in the Predefined Attribute object class. This field must be of the following form:

```
message file,set id,msg id,default text
```

The ordering of values must match the ordering in the **Predefined Attribute values** field.

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_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 reference:

"ODM Device Configuration Object Classes" on page 31

"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
Description

Device Type
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.

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.

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.

Descriptor **Description**

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.

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.

Descriptor	Description
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

Item	Description
/usr/lib/drivers directory	Contains device drivers.

Related reference:

"ODM Device Configuration Object Classes" on page 31

"loadext Device Configuration Subroutine" on page 23

"Writing a Define Method" on page 51

Related information:

odmadd command

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

Item	Description
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

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

Files

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

Related reference:

"getattr Device Configuration Subroutine" on page 20

Related information:

odm_initialize subroutine

Object Data Manager (ODM) Overview for Programmers

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

Item	Description
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

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

Files

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

Related reference:

"genmajor Device Configuration Subroutine" on page 16

"genminor Device Configuration Subroutine" on page 17

"Customized Device Driver (CuDvDr) Object Class" on page 11

Related information:

List of Device Configuration Subroutines

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

Item	Description
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

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

Files

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

Related reference:

"reldevno Device Configuration Subroutine" on page 43

"Customized Device Driver (CuDvDr) Object Class" on page 11

Related information:

List of Device Configuration Subroutines

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

Item	Description
-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.
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 reference:

"Writing an Unconfigure Method" on page 54

"Predefined Devices (PdDv) Object Class" on page 38

"Customized Attribute (CuAt) Object Class" on page 9

Related information:

Device Dependent Structure (DDS) Overview

Understanding Device Dependencies and Child Devices

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

Item	Description
-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 processing. 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.
   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" topic 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.
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

Item	Description
/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 reference:

"Customized Devices (CuDv) Object Class" on page 12

"Loading a Device Driver" on page 24

Related information:

Object Data Manager (ODM) Overview for Programmers

Understanding Device Dependencies and Child Devices

Configuration Manager Overview

System boot processing

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

Item	Description
-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.
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.
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.

Descriptor	Setting
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 reference:

"Predefined Devices (PdDv) Object Class" on page 38

"Predefined Attribute (PdAt) Object Class" on page 32

Related information:

Understanding Device Classes, Subclasses, and Types

Understanding Device Dependencies and Child Devices

Device location codes

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.

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

Item	Description
-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.

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

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

Related reference:

“Loading a Device Driver” on page 24

“Customized Devices (CuDv) Object Class” on page 12

“Predefined Devices (PdDv) Object Class” on page 38

Related information:

The Device Dependent Structure (DDS) Overview

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

Item	Description
-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" 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.
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

Item	Description
/dev directory	Contains the device special files.

Related reference:

"Customized Devices (CuDv) Object Class" on page 12

"Predefined Devices (PdDv) Object Class" on page 38

Related information:

Understanding Device Dependencies and Child Devices

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

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

Related reference:

"Writing an Unconfigure Method" on page 54

"Writing a Configure Method" on page 47

Communications Subsystem

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

Item	Description
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:

Item	Description
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 reference:

"ddwrite Communications PDH Entry Point" on page 82

"dd_fastwrt Communications PDH Entry Point" on page 74

Related information:

ddioctl subroutine

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.
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:

Item	Description
ENOMEM	Indicates no mbufs or mbuf clusters are available.
ENXIO	Indicates the adapter number is out of range.

Related reference:
"CIO_QUERY (Query Statistics) tsioctl PCI MPQP Device Handler Operation" on page 66
"MP_CHG_PARMS (Change Parameters) tsioctl PCI MPQP Device Handler Operation" on page 84
"tsioctl Multiprotocol (PCI MPQP) Device Handler Entry Point" on page 86

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

Item	Description
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:

Item	Description
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:

ddioctl subroutine

ddmpx subroutine

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:

Item **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_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:

Item	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

Item	Description
/usr/include/sys/comio.h	Contains the **session_blk** structure definition.

Related reference:

"tsselect Multiprotocol (PCI MPQP) Device Handler Entry Point" on page 92
"CIO_START (Start Device) tsioctl PCI MPQP Device Handler Operation" on page 69
"MP_CHG_PARMS (Change Parameters) tsioctl PCI MPQP Device Handler Operation" on page 84

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

Item	Description
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 reference:

"CIO_GET_STAT ddioctl Communications PDH Operation" on page 62

"CIO_START ddioctl Communications PDH Operation" on page 73

Related information:

ddioctl subroutine

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:

Item	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 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

Item	Description
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 reference:

"tsioctl Multiprotocol (PCI MPQP) Device Handler Entry Point" on page 86

"CIO_GET_STAT (Get Status) tsioctl PCI MPQP Device Handler Operation" on page 59

"MP_CHG_PARMS (Change Parameters) tsioctl PCI MPQP Device Handler Operation" on page 84

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

Item	Description
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:

ddioctl subroutine

ddmpx subroutine

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:

Item — **Description**

`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).

Item	Description
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:

Item	Description
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:

Item	Description
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:

Item	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

Item	Description
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 reference:
"tsioctl Multiprotocol (PCI MPQP) Device Handler Entry Point" on page 86
"CIO_GET_STAT (Get Status) tsioctl PCI MPQP Device Handler Operation" on page 59
"CIO_HALT (Halt Device) tsioctl PCI MPQP Device Handler Operation" on page 64

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

Item	Description
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 reference:

"CIO_GET_FASTWRT ddioctl Communications PDH Operation" on page 58

"ddread Communications PDH Entry Point" on page 79

"ddwrite Communications PDH Entry Point" on page 82

Related information:

ddioctl subroutine

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:

fp_ioctl subroutine

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

Item	Description
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:

Item	Description
ENXIO	Indicates an attempt to close an unconfigured device.

Related reference:

"CIO_HALT ddioctl Communications PDH Operation" on page 65

"CIO_START ddioctl Communications PDH Operation" on page 73

Related information:

ddopen subroutine

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

Item	Description
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 reference:

"ddclose Communications PDH Entry Point" on page 75

Related information:

ddmpx subroutine

Status Blocks for Communication Device Handlers Overview

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

Item	Description
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 reference:

"ddclose Communications PDH Entry Point" on page 75

"ddopen (Kernel Mode) Communications PDH Entry Point" on page 76

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

Item	Description
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 reference:

"CIO_START ddioctl Communications PDH Operation" on page 73

"ddwrite Communications PDH Entry Point" on page 82

Related information:

ddmpx subroutine

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

Item	Description
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:

ddmpx subroutine

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

Item	Description
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.

`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 reference:

"CIO_GET_FASTWRT ddioctl Communications PDH Operation" on page 58

Related information:

ddmpx subroutine

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:

Item	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_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 reference:

"tsioctl Multiprotocol (PCI MPQP) Device Handler Entry Point" on page 86

"CIO_HALT (Halt Device) tsioctl PCI MPQP Device Handler Operation" on page 64

"CIO_START (Start Device) tsioctl PCI MPQP Device Handler Operation" on page 69

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

Item	**Description**
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:

Item	**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 reference:

"tsopen Multiprotocol (PCI MPQP) Device Handler Entry Point" on page 89

"CIO_HALT (Halt Device) tsioctl PCI MPQP Device Handler Operation" on page 64

Related information:

close subroutine

fp_close subroutine

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

Item	Description
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 reference:

"tsioctl Multiprotocol (PCI MPQP) Device Handler Entry Point"

Related information:

ddconfig subroutine

PCI MPQP Device Handler Interface Overview

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:

Item	Description
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

Item	Description
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:

Item	Description
ENOMEM	Indicates the no memory buffers (mbufs) or mbuf clusters are available.
ENXIO	Indicates the adapter number is out of range.

Related reference:

"tsconfig Multiprotocol (PCI MPQP) Device Handler Entry Point" on page 86

"CIO_HALT (Halt Device) tsioctl PCI MPQP Device Handler Operation" on page 64

"MP_CHG_PARMS (Change Parameters) tsioctl PCI MPQP Device Handler Operation" on page 84

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

Item **Description**

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:

Item	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 reference:

"tsclose Multiprotocol (PCI MPQP) Device Handler Entry Point" on page 84

"tsconfig Multiprotocol (PCI MPQP) Device Handler Entry Point" on page 86

Related information:

ddmpx subroutine

Communications I/O Subsystem: Programming Introduction

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

Item	Description
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:

Item	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 reference:

“tsclose Multiprotocol (PCI MPQP) Device Handler Entry Point” on page 84

“tsconfig Multiprotocol (PCI MPQP) Device Handler Entry Point” on page 86

“CIO_START (Start Device) tsioctl PCI MPQP Device Handler Operation” on page 69

Related information:

fp_open subroutine

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:

Item	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.

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

Item	Description
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:

Item	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 reference:

"tswrite Multiprotocol (PCI MPQP) Device Handler Entry Point" on page 93

Related information:

read or readx

Communications Physical Device Handler Model Overview

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

Item	Description
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:

Item	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 reference:

"ddselect Communications PDH Entry Point" on page 81

Related information:

poll subroutine

select subroutine

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:

Item — **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.

Parameters

Item	Description
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:

Item	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 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 reference:

“tsread Multiprotocol (PCI MPQP) Device Handler Entry Point” on page 90

“CIO_GET_STAT (Get Status) tsioctl PCI MPQP Device Handler Operation” on page 59

Related information:

write or writex

Binary Synchronous Communication (BSC) with the MPQP Adapter

LFT Subsystem

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

Item	Description
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

Item	Description
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.

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

Item	Description
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.

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

Item	Description
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

Item	Description
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

Item	Description
FileDescriptor	Specifies the open file descriptor for the dials.
Arg	Specifies the address of the **dialsetgrand** structure.

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

Item	Description
FileDescriptor	Specifies the open file descriptor for the gio adapter.
Arg	Specifies the address of a **gioqueryid** structure.

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.
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

Item	Description
FileDescriptor	Specifies the open file descriptor for the device.
Arg	Specifies the address of the **devinfo** structure.

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

Item	**Description**
FileDescriptor	Specifies the open file descriptor for the keyboard.
Arg	Specifies the address of the **KSALARM** structure.

Related reference:

"KSVOLUME (Set Alarm Volume) ioctl" on page 108

Related information:

chhwkbd subroutine

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

Item	**Description**
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.*/
```

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

Item	Description
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.

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

Item	Description
FileDescriptor	Specifies the open file descriptor for the keyboard.
Arg	Specifies the address of an array of characters or is equal to NULL.

Related reference:
"KSKAPACK (Acknowledge Keep Alive Poll)"

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

Item	Description
FileDescriptor	Specifies the open file descriptor for the keyboard.

Related reference:
"KSKAP (Enable/Disable Keep Alive Poll)" on page 102

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

Item	Description

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.

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

Item	Description
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

Item	**Description**
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

Item	Description
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

Item	Description
FileDescriptor	Specifies the open file descriptor for the keyboard.

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

Item	Description
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:

chhwkbd subroutine

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

Item	Description
FileDescriptor	Specifies the open file descriptor for the keyboard.
Arg	Specifies the address of an integer that contains the desired typematic rate.

Related information:

chhwkbd subroutine

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

Item **Description**

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.*/
```

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;
```

Structure Member **Description**

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.

Related reference:

"lft_t Structure"

"vtmstruct Structure" on page 127

"phys_displays Structure" on page 117

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 reference:

"lft_dds_t Structure" on page 109

"vtmstruct Structure" on page 127

"phys_displays Structure" on page 117

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

Item	Description
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).

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

Item	Description
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

Item	Description
FileDescriptor	Specifies the open file descriptor.

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

Item	**Description**
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

Item	**Description**
FileDescriptor	Specifies the open file descriptor for the mouse.
Arg	Specifies the address of an **URERING** structure.

MREGRINGEXT (Register Extended Input Ring)

Purpose

Registers extended input ring.

Syntax

```
#include <sys/inputdd.h>
int ioctl (FileDescriptor, MREGRINGEXT, arg);
int FileDescriptor;
struct uregring *arg;
```

Description

This function enqueues the extended mouse event reports onto the input ring. Extended reports contain additional information such as mouse wheel movement. This ioctl operation has the same parameters and is processed in the same manner as the `MREGRING` ioctl function.

Parameters

Item	Description
FileDescriptor	Specifies the open file descriptor for the mouse.
Arg	Specifies the address of the UREGRING structure.

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

Item | **Description**

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     */
```

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

Item	Description
FileDescriptor	Specifies the open file descriptor for the mouse.

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

Item	Description
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    */
```

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

Item	Description
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*/
```

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

Item	Description
FileDescriptor	Specifies the open file descriptor for the mouse.
Arg	Specifies the address of the desired threshold.

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:

```
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. clean up */
#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 reference:

"lft_t Structure" on page 111

"lft_dds_t Structure" on page 109

"vtmstruct Structure" on page 127

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

Item	**Description**
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 reference:

"TABRESOLUTION (Set Tablet Resolution)" on page 123

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

Item	Description
FileDescriptor	Specifies the open file descriptor for the tablet.
Arg	Specifies the address of the dead zone specification.

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

Item	Description
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               */`

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

Item	Description
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

Item	Description
FileDescriptor	Specifies the open file descriptor for the tablet.
Arg	Specifies the address of a **uregring** structure.

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

Item	Description
FileDescriptor	Specifies the open file descriptor for the tablet.
Arg	Specifies the address of an integer that contains the desired resoultion.

Related reference:

"TABCONVERSION (Set Tablet Conversion Mode)" on page 121

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

Item	Description
FileDescriptor	Specifies the open file descriptor for the tablet.

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

Item	Description
FileDescriptor	Specifies the open file descriptor for the tablet.
Arg	Specifies the address of an integer that contains the desired sample rate.

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

Item **Description**

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.

Item **Description**

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.

Item	Description
	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.

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.

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.

Structure Member **Description**

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.

Related reference:

"lft_t Structure" on page 111

"lft_dds_t Structure" on page 109

"phys_displays Structure" on page 117

Printer Subsystems

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 reference:

"piocmdout Subroutine"

Related information:

lineout subroutine

Adding a New Printer Type to Your System

Example of Print 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

Item	Description
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 reference:

"piogetvals Subroutine" on page 135

Related information:

lineout subroutine

Adding a New Printer Type to Your System

Print formatter example

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

Item	Description
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

Adding a New Printer Type to Your System

Print formatter example

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

Item	Description
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

Item	Description
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

Item	Description
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 reference:

"piogetvals Subroutine" on page 135

Related information:

Adding a New Printer Type to Your System

Print formatter example

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

Item	Description
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

Item	Description
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

Item	Description
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 reference:

"piogetvals Subroutine"

Related information:

Adding a New Printer Type to Your System

Print formatter example

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

Item	Description
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 reference:

"piocmdout Subroutine" on page 129

Related information:

atoi subroutine

Adding a New Printer Type to Your System

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

Item	Description
msgstr	Points to the string of message text to be sent.

Related information:

Understanding Embedded References in Printer Attribute Strings

Adding a New Printer Type to Your System

Print formatter example

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

Item	Description
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

Item	Description
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

Item	Description
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

Item	Description
1	Indicates that the **pioputstatus** subroutine was successful.
-1	Indicates that an error occurred.

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 reference:

"passthru Subroutine" on page 128

Related information:

initialize subroutine

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

Item	Description
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 reference:

"piocmdout Subroutine" on page 129

Related information:

qdaemon command

Adding a New Printer Type to Your System

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.

Related reference:

"Subroutines for Writing a Print Formatter"

Related information:

Printer Addition Management Subsystem: Programming Overview

Adding a New Printer Type to Your System

Print formatter example

Subroutines for Writing a Print Formatter

The **pioformat** formatter driver requires a print formatter to contain the following function routines:

Item	Description
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.

Related reference:

"Subroutines for Print Formatters"

Related information:

Printer Addition Management Subsystem: Programming Overview

Adding a New Printer Type to Your System

Print formatter example

SCSI Subsystem

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

Item	Description
/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related reference:
"tape SCSI Device Driver" on page 215
"scdisk SCSI Device Driver" on page 151
"Parallel SCSI Adapter Device Driver"

Parallel 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.
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.

Value	Description
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

SCSI_ERR1:

Field	**Description**
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.
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.

SCSI_ERR3:

Field	Description
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.
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.
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
`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

Item	Description
/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 reference:

"SCIOCMD SCSI Adapter Device Driver ioctl Operation" on page 169

"scdisk SCSI Device Driver"

"tape SCSI Device Driver" on page 215

"tmscsi SCSI Device Driver" on page 227

scdisk SCSI Device Driver

Purpose

Supports the small computer system interface (SCSI) hard disk, CD-ROM (compact-disc 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 hard disk, CD-ROM, and read/write optical drive operations are implemented by 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 that is based on the ODM **reserve_policy** attribute. In the past, the **open** subroutine always applied an SCSI2 reserve. The **open** and **close** subroutines support working with multiple paths to a device if the device is an 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 that is 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:

Item	Description
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 process 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_LUN	On a device that supports Lun Level Reset, the device is reset regardless of any reservation that is placed by another initiator before the open sequence takes place. If the device does not support Lun Level Reset, and both **SC_FORCED_OPEN_LUN** and **SC_FORCE_OPEN** flags are set, then a target reset occurs before the open sequence takes place.
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.
SC_PR_SHARED_REGISTER	In a multi-initiator shared device environment, a Persistent Reserve with service action `Register and Ignore Key` is sent to the device as part of the open sequence. This feature is aimed at the cluster environment, where an upper management software needs to follow an advisory lock mechanism to control when the initiator reads or writes. The device is required to support Persistent Reserve (refer to SCSI Primary Command version 2 description of Persistent Reserve).

SCSI Options to the openx Subroutine in *Kernel Extensions and Device Support Programming Concepts* gives more specific information about 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:

Item	Description
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:

- Hard disk and read/write optical devices only

- CD-ROM devices only
- Hard disk, CD-ROM, and read/write optical devices

Hard disk and read/write optical devices

The following **ioctl** operations are available for the hard disk and read/write optical devices:

Item — **Description**

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 when it is 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.

DKLOGSENSE

Provides a means to issue the **LOG SENSE** command on devices that is successfully opened. Any application that issues this **ioctl** operation must pass the address of the **struct sc_log_sense** (defined in **/usr/include/sys/scsi.h**) and the structure is filled as follows:

1. **page_code** and **subpage_code** for the requested LOG Page
2. **pc** set to the value **CUMUL_VAL**.
3. **allocation_length** – If this field is set to zero, only the log page header that consists of the **page code** and the log page length is returned. If this field is nonzero, it must equal the length of the log page excluding the log page header of size 4 bytes. If the user specifies an allocation length less than the actual log page length, then only the requested length of log data is returned.
4. **log_data** contains the allocated memory address for storing the data that is returned from the **ioctl** operation.

If the requested log page is **SCSI_BSR_LOG_PAGE** (defined in **/usr/include/sys/scsi.h**) then the **log_data** points to the **struct sc_bsr_log_data** allocated by the caller. The caller also allocates the memory for the **struct sc_bms_log_data** such that total of **sizeof struct sc_bsr_log_data** and memory allocated for the **struct sc_bms_log_data** is equal to the **allocation_length**.

Otherwise (for log pages other than **SCSI_BSR_LOG_PAGE**), it points to a chunk of memory equal to **allocation_length**.

Following is the example code for filling the **sc_log_sense** structure:

```
struct sc_log_sense log_sense;

melog_sense, '\0', sizeof(struct sc_log_sense));
log_sense.page_code = SCSI_BSR_LOG_PAGE ;
log_sense.subpage_code = 0;
log_sense.pc = CUMUL_VAL;
log_sense.allocation_length = 16;
if (log_sense.allocation_length)
 {
 if (log_sense.page_code == SCSI_BSR_LOG_PAGE)
  {
   log_sense.log_data = (struct sc_bsr_log_data *) malloc(sizeof(struct sc_bsr_log_data));
  }
  else
  {
   log_sense.log_data = (char *) malloc(log_sense.allocation_length);
  }
  if (log_sense.log_data == NULL)
  exit(-1);
  if (log_sense.page_code == SCSI_BSR_LOG_PAGE)
  {
   bms_param_len = log_sense.allocation_length - sizeof(struct sc_scan_status);
   ((struct sc_bsr_log_data *)(log_sense.log_data))-> bms_log_data = (struct sc_bms_log_data *)
  malloc(bms_param_len);
  if (((struct sc_bsr_log_data *)(log_sense.log_data))-> bms_log_data == NULL)
  exit(-1);
  }
 }
 rc = ioctl(fd, DKLOGSENSE,&log_sense);
```

Item **Description**

The **DKLOGSENSE ioctl** operation returns the following data by using the struct **sc_log_sense** (**rc=0** indicates success):

1. **returned_length** field contains the length of the bytes requested or zero if the user specified an allocation length of zero.
2. **adapter_status**, **scsi_status**, **sense_key**, **scsi_asc**, **scsi_ascq** set with the error return status for the **LOG SENSE** command.
3. **log_data** field points to the memory containing the data returned by the **LOG SENSE** command. Driver will parse and fill fields for the **struct sc_bsr_log_data** and **struct bms_log_data** for the **Background Scan Results log page**. The **ioctl** caller prints structure fields to view the data. Otherwise, this memory is a **char *** to the log data of **returned_length**. The data excludes the log page header.

```
if (log_sense.page_code == SCSI_BSR_LOG_PAGE)
{
    bms_cnt = (log_sense.returned_length - 16)/24;
    printf("Background Scan Results Log Page:\n");
    printf("Scanning Status Parameter:\n");
    bsr_log_data = (struct sc_bsr_log_data *)log_sense.log_data;
    scan_status  = &(bsr_log_data->scan_status);
    printf("Parameter Code \t:\t %x\n", scan_status->param_code);
    printf("Parameter Control Byte \t:\t %x",scan_status->param_ctrl_bits);
    printf("Parameter Length \t:\t %x\n", scan_status->param_length);
    printf("Power Age \t:\t %x\n", scan_status->power_age);
    printf("Scan Status \t:\t %x\n", scan_status->scan_status);
    printf("Scan Count \t:\t %x\n", scan_status->scan_count);
    printf("Scan Progress \t:\t %x \n", scan_status->scan_progress);
    printf("BMS Count \t:\t %x\n", scan_status->bms_count);
    printf("Background Medium Scan Parameter for %d Elements:\n",bms_cnt);

    for (i=0; <ibms_cnt; i++)
    {
        bms_data = bsr_log_data->bms_log_data;
        printf("Parameter Code \t:\t %x\n", bms_data->param_code);
        printf("Parameter Control Byte \t:\t%x",bms_data->param_ctrl_bits);
        printf("Parameter Length \t:\t %x\n", bms_data->param_length);
        printf("Power Age \t:\t %x\n", bms_data->power_age);
        printf("SenseKey & Reassign Status \t:\t %x\n",bms_data->status_snskey);
        printf("ASC \t:\t %x\n", bms_data->asc);
        printf("ASCQ \t:\t %x\n", bms_data->ascq);

        for (j=0; j<5; j++)
            printf("vendor_data[%d] \t:\t%x\n",j,bms_data->vendor_data[j]);

        printf("LBA \t:\t %llx\n", bms_data->lba);
    }

} else {
    Log data received is a char buffer of 'returned_length' size.
    So print the data byte by byte.
}
```

Item **Description**

DKLOGSELECT

Provides a means to issue the **LOG SELECT** command.

Any application that issues the **DKLOGSELECT ioctl** operation is expected to pass the address of the **DKLOGSELECT** (defined in **/usr/include/sys/scsi.h**) filled as follows:

1. **page_code** and **subpage_code** for the requested LOG Page
2. **pcr**, **sp**, **pc**, and **param_length** as per the SCSI Primary Commands Standard Version 4 (SPC4) requirements.
3. **log_data** points to the memory that contains the parameters that must be sent to the **LOG SELECT** command.

Following is an example for filling the **sc_log_select** structure to clear the **SCSI_BSR_LOG_PAGE data****.

```
struct sc_log_select log_select;

memset(&log_select, '\0', sizeof(struct sc_log_select));
log_select.page_code = SCSI_BSR_LOG_PAGE;
log_select.subpage_code = 0;
log_select.pcr = 1;
log_select.sp = 0;
log_select.pc = CUMUL_VAL;
log_select.param_length = 0;
if (log_select.param_length)
{
 log_select.log_data = (char *)malloc(log_select.param_length);
 if (log_select.log_data == NULL) exit(-1);
}
rc = ioctl(fd, DKLOGSELECT,&log_select);
```

This **ioctl** operation returns the following data by using the **struct sc_log_select** (**rc=0** indicates success) **adapter_status**,**scsi_status**, **sense_key**, **scsi_asc**, and **scsi_ascq** fields reporting the error completion status of the **LOG SELECT** command.

CD-ROM Devices Only

The following **ioctl** operation is available for CD-ROM devices only:

Item **Description**

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.

Hard disk, CD-ROM, and read/write optical devices

The following **ioctl** operations are available for hard disk, CD-ROM, and read/write optical devices:

Item **Description**

IOCINFO

Returns the **devinfo** structure that is defined in the **/usr/include/sys/devinfo.h** file. The **IOCINFO** operation is the only operation that is defined for all device drivers that use the **ioctl** subroutine. The remaining operations are all specific to hard disk, CD-ROM, and read/write optical devices.

Item **Description**

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 when it is 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 might be substituted for the **DKIORDSE** operation when you issue a **read** command to and obtain sense data from a CD-ROM device. **DKIORDSE** is the recommended operation.

DKIOCMD

When the device is 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 must evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions must 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 is 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 hard 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 is successfully opened. As a result media can be ejected by 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 hard 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 is 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 hard 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**.

Item **Description**

DKFORMAT
Issues a format unit command to the specified device when the device is 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. It 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 3 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 hard 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 must evaluate the returned status bytes to determine why the operation failed and what recovery actions must be taken.

Item **Description**

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 that is 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 is 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 is successfully opened in the exclusive access mode.

If a CD-ROM is not configured for different data modes (through 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 also results 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 is 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 must evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions must be taken.

Item **Description**

If a **DK_PASSTHRU** operation fails because a field in the **sc_passthru** structure has an invalid value, the subroutine returns a value of -1 and set the errno global variable to **EINVAL**. The **einval_arg** field is 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 that no additional information on the failure is available.

DK_PASSTHRU operations are further subdivided into requests which quiesce other I/O requests before issuing the request and requests that do not quiesce I/O requests. 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 version field of the **sc_passthru** structure can be set to the value of SC_VERSION_2, and the user can provide the following fields:

- **variable_cdb_ptr** is a pointer to a buffer that contains the Variable SCSI cdb.
- **variable_cdb_length** determines the length of the *cdb* variable to which the **variable_cdb_ptr** field points.

On completion of the **DK_PASSTHRU** ioctl request, the **residual** field indicates the leftover data that device did not fully satisfy for this request. On a successful completion, the **residual** field would indicate that the device does not have the all data that is requested or the device has less than the amount of data that is requested. On a failure completion, the user must check the **status_validity** field to determine whether a valid SCSI bus problem exists. In this case, the **residual** field would indicate the number bytes that the device failed to complete for this request.

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.

DKPRES_READKEYS

When the device is successfully opened, the **DKPRES_READKEYS** operation provides a means to read the Persistent Reserve Registration Keys on the device. The *arg* parameter to the **DKPRES_READKEYS** contains the address of the **dk_pres_in** structure. This structure is defined in **/usr/include/sys/scdisk.h**. The user must provide a buffer area and size for the registered keys to be returned. The *returned_length* variable sets the number of bytes returned.

In a shared-access or clustered environment, this operation identifies all registered keys for a particular lun.

Note: For the **DKPRES_READKEYS** operation and following Persistent Reserve related operation, the interpretation of the returned value and scsi status is as follows:

- On successful attempt of the call, a 0 is returned.
- After a call fails, a -1 is returned and the **errno** global variable is set. For a specific description of the **errno** value, refer to **/usr/include/erno.h**. In addition, the SCSI status, along with the Sense Code, ASC and ASCQ, is set to provide further information about why the command failed. Refer to SCSI Specification on the interpretation of the SCSI status failure code.

Item **Description**

DKPRES_READRES

When the device is successfully opened, the **DKPRES_READRES** operation provides a means to read the Persistent Reserve Reservation Keys on the device. The *arg* parameter to the **DKPRES_READKEYS** contains the address of the **dk_pres_in** structure. This structure is defined in **/usr/include/sys/scdisk.h**. The user must provide a buffer area and size for the reservation information to be returned. The *returned_length* variable sets the number of bytes returned. In a shared-access or clustered environment, this operation identifies the primary initiator that holds the reservation.

DKPRES_CLEAR

When the device is successfully opened, the **DKPRES_CLEAR** operation provides a means to clear all Persistent Reserve Reservation Keys and Registration Keys on the device. The *arg* parameter to **DKPRES_CLEAR** contains the address of the **dk_pres_clear** structure. This structure is defined in **/usr/include/sys/scdisk.h**.

Attention: **Attention:** Exercise care when issuing the **DKPRES_CLEAR** operation. This operation leaves the device unreserved, which could allow a foreign initiator to access the device.

DKPRES_PREEMPT

When the device is successfully opened, the **DKPRES_PREEMPT** operation provides a means to preempt a Persistent Reserve Reservation Key or Registration Key on the device. The *arg* parameter to the **DKPRES_PREEMPT** contains the address of the **dk_pres_preempt** structure. This structure is defined in **/usr/include/sys/scdisk.h**. The user must provide the second party initiator key on the device to be preempted. If the second party initiator holds the reservation to the device, then the initiator that issues the preemption becomes the owner of the reservation. Otherwise, the second party initiator access is revoked.

In order for this operation to succeed, the initiator must be registered with the device first before any preemption can occur. In a shared-access or clustered environment, this operation is used to preempt any operative or inoperative initiator, or any initiator that is not recognized to be part of the shared group.

DKPRES_PREEMPT_ABORT

This operation is the same as the **DKPRES_PREEMPT**, except the device follows the SCSI Primary Command Specification in aborting tasks that belong to the preempted initiator.

DKPRES_REGISTER

When the device is successfully opened, the **DKPRES_REGISTER** operation provides a means to register a Key with the device. The Key is extracted from ODM Customize Attribute and passed to the device driver during configuration. The *arg* parameter to the **DKPRES_REGISTER** contains the address of the **dk_pres_register** structure. This structure is defined in **/usr/include/sys/scdisk.h**.

In a shared-access or clustered environment, this operation attempts a registration with the device, then follows with a read reservation to determine whether the device is reserved. If the device is not reserved, then a reservation is placed with the device.

DK_RWBUFFER

When the device is 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 performs 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 with the **DK_PASSTHRU ioctl** operation, 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 must be set according to the required 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 quiesces all other I/O requests from the initiator that issues the Write Buffer ioctl until the entire operation completes. Once the write buffer ioctl completes, all quiesced I/O requests are resumed.

Item	Description
	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 must evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions must be taken. If a **DK_RWBUFFER** operation fails because a field in the **sc_rwbuffer** structure has an invalid value, the subroutine returns 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. It is useful for applications such as microcode download where the user provides a pointer to the entire microcode image, but, because of size restrictions of the device buffers, 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 is 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 is issued to the device. The number of Write Buffer commands (SCSI CDBs) issued is calculated by dividing the **data_length** by the required **fragment_size**. This value is incremented by 1 if the **data_length** is not an even multiple of **fragment_size**, and the final data transfer is the size of this residual amount. For each Write Buffer command that is issued, the **buffer_offset** is set to the value provided in the **sc_rwbuffer** struct (microcode downloads to SCSD devices requires this value to be set to 0). For the first command issued, the **buffer_ID** is set to the value provided in the **sc_rwbuffer** struct. For each subsequent Write Buffer command that is issued, the **buffer_ID** is incremented by 1 until all fragments are sent. Writing to noncontiguous **buffer_ID**s through a single **DK_RWBUFFER** ioctl is not supported. If this functionality is wanted, multiple **DK_RWBUFFER** ioctls must be issued with the **buffer_ID** set appropriately for each invocation. **Note:** No I/O request is quiesced between ioctl invocations. If **fragment_size** is set to zero, an **errno** of **EINVAL** is returned. If the desire is to send the entire buffer with one SCSI Write buffer command, this field must be set equal to **data_length**. An error of **EINVAL** is also returned if the **fragment_size** is greater than the **data_length**. The Parameter List Length (**fragment_size**) plus the Buffer Offset cannot 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 results in a SCSI error at the device, and the Write Buffer ioctl reports 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 must 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 overrides the **DKPATHFORCE** option and send the I/O command down the path that is specified in **scdisk_pathiocmd** structure.

Item	Description
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 that is 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 that is 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 hard disk, CD-ROM, and read/write optical drives have the following hardware requirements:

- SCSI hard 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 hard disks and read/write optical drives.
- SCSI hard 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 hard disk or read/write optical drive does not support it, the system error log might indicate premature drive failure.
- SCSI CD-ROM and read/write optical drives must support the 10-byte SCSI read command.
- SCSI hard 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 by 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 by 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 subchannel
  - pause resume
  - play audio MSF
  - play audio track index
  - read TOC

Error Conditions

Possible **errno** values for **ioctl**, **open**, **read**, and **write** subroutines when you use the **scdisk** device driver include:

Item **Description**

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 invalid user address.

EFORMAT
Indicates that the target device has unformatted media or media in an incompatible format.

EINPROGRESS
Indicates that 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 the allowed length 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 incorrect configuration command is 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 that are 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 indicated an unrecoverable hardware error.

EMEDIA
Indicates one of the following circumstances:

- The target device indicated an unrecoverable media error.
- The media was changed.

EMFILE
Indicates that 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 that no media is in the drive.

Item	Description
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 hard disk.
EPERM	Indicates that the attempted subroutine requires appropriate authority.
ESTALE	Indicates that 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 exceeded the specified timer value.
EWRPROTECT	Indicates one of the following circumstances: • An **open** operation that requested the **read/write** mode was attempted on read-only media. • A **write** operation was attempted to read-only media.

Reliability and Serviceability Information

SCSI hard disk devices, CD-ROM drives, and read/write optical drives return the following errors:

Item	Description
ABORTED COMMAND	Indicates that the device ended the command
ADAPTER ERRORS	Indicates that the adapter returned an error
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 incorect command or command parameter
MEDIUM ERROR	Indicates that the command ended with an unrecoverable media error condition
NOT READY	Indicates that the logical unit is offline or media is missing
RECOVERED ERROR	Indicates that the command was successful after some recovery was applied
UNIT ATTENTION	Indicates that the device is reset or the power is turned on

Error Record Values for Media Errors

The fields that are defined in the error record template for hard disk, CD-ROM, and read/write optical media errors are:

Item	Description
`Comment`	Indicates hard 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 must be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error log entry must 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 must be performed • 1601, which indicates the removable media must 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 must be performed • 1601, which indicates the removable media must be replaced and tried again

Item **Description**

`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 that are defined in the error record template for hard disk, CD-ROM, and read/write optical hardware errors, as well as hard-aborted command errors are:

Item **Description**

`Comment` Indicates hard 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 must be included when an error report is generated.

`Log` Equals a value of True, which indicates an error log entry must 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 must 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 that are defined in the error record template for hard disk, CD-ROM, and read/write optical media errors adapter-detected hardware errors are:

Item	Description
`Comment`	Indicates adapter-detected hard 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 must be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error-log entry must 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 must 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 that are defined in the error record template for hard disk, CD-ROM, and read/write optical media errors recovered errors are:

Item **Description**

`Comment` Indicates hard 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 must be included when an error report is generated.

`Log` Equals a value of True, which indicates an error log entry must 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 must be performed
- 1601, which indicates the removable media must be replaced and tried again

`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 must be performed
- 1601, which indicates the removable media must be replaced and tried again

`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 that are defined in the error record template for hard disk, CD-ROM, and read/write optical media errors unknown errors are:

Item **Description**

`Comment`
Indicates hard 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 must be included when an error report is generated.

`Log`
Equals a value of True, which indicates an error log entry must 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 must be performed
- 1601, which indicates the removable media must be replaced and tried again

`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) occurs if devices that support paging, logical volumes, or mounted file systems are accessed by 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 must not be used for other purposes.

The special files that are used by the **scdisk** device driver include the following (listed by type of device):

- Hard disk devices:

Item	Description
/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 hard disks.
/dev/hdisk0, **/dev/hdisk1**,..., **/dev/hdisk***n*	Provides an interface to allow SCSI device drivers block I/O access to SCSI hard disks.

- CD-ROM devices:

Item	Description
/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:

Item	Description
/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 that the drive is accessed as a raw device rather than a block device. Performing raw I/O with a hard 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 reference:

"Parallel SCSI Adapter Device Driver" on page 143

Related information:

Special Files Overview

A Typical Initiator-Mode SCSI Driver Transaction Sequence

ioctl or ioctlx

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 it happens, the caller must evaluate the returned status bytes to determine why the operation failed and what recovery actions must be taken.

If the **SCIOCMD** operation fails because a field in the **sc_passthru** structure has an invalid value, the subroutine returns a value of -1, the errno global variable is set to **EINVAL**, and the **einval_arg** field is 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 that no additional information is available.

The version field of the **sc_passthru** structure can be set to the value of SC_VERSION_2 in the **/usr/include/sys/scsi.h** file, and the user can provide the following fields:

- **variable_cdb_ptr** is a pointer to a buffer that contains the Variable SCSI cdb.
- **variable_cdb_length** determines the length of the *cdb* variable to which the **variable_cdb_ptr** field points.

On completion of the **SCIOCMD** ioctl request, the **residual** field indicates the leftover data that device did not fully satisfy for this request. On a successful completion, the **residual** field would indicate that the device does not have the all data that is requested or the device has less than the amount of data that is requested. On a failure completion, the user must check the **status_validity** field to determine whether a valid SCSI bus problem exists. In this case, the **residual** field would indicate the number bytes that the device failed to complete for this request.

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:

Item	Description
EIO	A system error occurred. Consider trying the operation several (three) times because another attempt might 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 must be inspected. If the **status_validity** field is zero and remains so on successive trials, an unrecoverable error occurred. If the **status_validity** field is **SC_SCSI_ERROR** and the **scsi_status** field contains a *Check Condition* status, a SCSI request sense must be issued by using the **SCIOCMD** ioctl to recover the sense data.
EFAULT	A user process copy failed.
EINVAL	The device is not opened, or the caller set a field in the **sc_passthru** structure to an invalid value.
EACCES	The adapter is in diagnostics mode.
ENOMEM	A memory request failed.
ETIMEDOUT	The command timed out. Consider trying the operation several times because another attempt might be successful.
ENODEV	The device is not responding.
ETIMEDOUT	The operation did not complete before the timeout value was exceeded.

Files

Item	Description
/dev/scsi0, /dev/scsi1, ... /dev/scsin	Provides an interface for all SCSI device drivers to access SCSI devices or adapters.

Related reference:
"Parallel SCSI Adapter Device Driver" on page 143

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.
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

Item	Description
/dev/scsi0, **/dev/scsi1**,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices/adapters.

Related reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/dev/scsi0, **/dev/scsi1**,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices and adapters.

Related reference:

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

Related information:

SCSI Subsystem Overview

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

Item	Description
/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 reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/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 reference:

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/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 reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/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 reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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.
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

Item	Description
/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 reference:
"tape SCSI Device Driver" on page 215
"scdisk SCSI Device Driver" on page 151
"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/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 reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/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 reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/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 reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

"tmscsi SCSI Device Driver" on page 227

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

Item	Description
/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 reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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:

Item	Description
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

Item	Description
/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 reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

"tmscsi SCSI Device Driver" on page 227

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.
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

Item	Description
/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 reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/dev/scsi0, **/dev/scsi1**,..., **/dev/scsi***n*	Provide an interface to allow SCSI device drivers to access SCSI devices or adapters.

Related reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/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 reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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.
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

Item	Description
/dev/ses0,/dev/ses1...,/dev/sesn	Provides an interface to allow SCSI device drivers access to SCSI enclosure services devices.

Related reference:

"Parallel SCSI Adapter Device Driver" on page 143

Related information:

SCSI Subsystem Overview

Understanding the sc_buf Structure

scsidisk SAM Device Driver

Purpose

Supports the Fibre Channel Protocol for SCSI (FCP), Serial Attached SCSI (SAS), and the SCSI protocol over Internet (iSCSI) hard 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 hard disk, CD-ROM, and read/write optical drive operations are implemented by 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 that is based on the ODM **reserve_policy** attribute, previously the **open** subroutine always applied an SCSI2 reserve. The **open** and **close** subroutines support working with multiple paths to a device if the device is an 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 that is 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:

Item	Description
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_LUN	On a device that supports Lun Level Reset, the device is reset regardless of any reservation that is placed by another initiator before the open sequence takes place. If the device does not support Lun Level Reset, and both **SC_FORCED_OPEN_LUN** and **SC_FORCE_OPEN** flags are set, then a target reset occurs before the open sequence takes place.
SC_FORCED_OPEN	Initiates actions during the **open** operation to break any reservation that might exist on the device. This action might include a target reset. **Note:** A target reset resets 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.

Item	Description
SC_PR_SHARED_REGISTER	In a multi-initiator shared device environment, a Persistent Reserve with service action `Register and Ignore Key` is sent to the device as part of the open sequence. This feature is aimed at the cluster environment, where an upper management software must follow an advisory lock mechanism to control when the initiator reads or writes. The device is required to support Persistent Reserve (refer to SCSI Primary Command version 2 description of Persistent Reserve).

Options to the openx Subroutine in *Kernel Extensions and Device Support Programming Concepts* gives more specific information about the **open** operations.

readx and writex Subroutines

The **readx** and **writex** subroutines provide additional parameters that 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:

Item	Description
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:

- Hard disk and read/write optical devices only
- CD-ROM devices only
- Hard disk, CD-ROM, and read/write optical devices

Hard disk and read/write optical devices

The following **ioctl** operation is available for hard disk and read/write optical devices only:

Item	Description
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 when it operates 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.

Hard disk, CD-ROM, and read/write optical devices

The following **ioctl** operations are available for hard disk, CD-ROM, and read/write optical devices:

Item **Description**

IOCINFO

Returns the **devinfo** structure that is 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 are all specific to hard 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 when it operates 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 is 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 must evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions must be taken.

The version field of the **scsi_iocmd** structure can be set to the value of SCSI_VERSION_2, and the user can provide the following fields:

- **variable_cdb_ptr** is a pointer to a buffer that contains the Variable SCSI cdb.
- **variable_cdb_length** determines the length of the *cdb* variable to which the **variable_cdb_ptr** field points.

On completion of the **DKIOLCMD** ioctl request, the **residual** field indicates that the leftover data that device did not fully satisfy for this request. On a successful completion, the **residual** field would indicate that the device does not have the all data that is requested or the device has less than the amount of data that is requested. On a failure completion, the user must check the **status_validity** field to determine if a valid SCSI bus problem exists. In this case, the **residual** field would indicate the number bytes that the device failed to complete for this request.

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 is 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 hard 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 is successfully opened. As a result media can be ejected by 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 hard 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**.

Item **Description**

DKEJECT
Issues an eject media command to the drive when the device is 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 hard 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 is 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. It 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 hard 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 must evaluate the returned status bytes to determine why the operation failed and what recovery actions must be taken.

Item **Description**

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 that is 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 is 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 is successfully opened in the Exclusive Access mode.

If a CD-ROM is not configured for different data modes (through 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 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**.

Item **Description**

DK_PASSTHRU

When the device is successfully opened, **DK_PASSTHRU** provides the means for issuing any SCSI command to the specified device. The device driver performs 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**. As a result, **DK_PASSTHRU** 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 it happens the caller must evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions must be taken.

If a **DK_PASSTHRU** operation fails because a field in the **sc_passthru** structure has an invalid value, the subroutine returns a value of -1 and sets the errno global variable to **EINVAL**. The **einval_arg** field is 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 that no additional information about the failure is available.

DK_PASSTHRU operations are further subdivided into requests which quiesce other I/O before 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 is mixed with other I/O requests. **SC_MIX_IO** requests that write data to devices are prohibited and fail. When it 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 are 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 fails with a return code of -1, the errno global variable is set to **EINVAL**, and the **einval_arg** field is 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 fails with a return code of -1, the errno global variable is set to **EINVAL**, the **einval_arg** field is set to a value of **SC_PASSTHRU_INV_TO** (defined in the **/usr/include/sys/scsi.h** file), and the **timeout_value** is set to the largest allowed value.

The version field of the **sc_passthru** structure can be set to the value of SCSI_VERSION_2, and the user can provide the following fields:

- **variable_cdb_ptr** is a pointer to a buffer that contains the Variable SCSI cdb.
- **variable_cdb_length** determines the length of the *cdb* variable to which the **variable_cdb_ptr** field points.

On completion of the **DK_PASSTHRU** ioctl request, the **residual** field indicates that the leftover data that device did not fully satisfy for this request. On a successful completion, the **residual** field would indicate the device does not have the all data that is requested or the device has less than the amount of data that is requested. On a failure completion, the user must check the **status_validity** field to determine whether a valid SCSI bus problem exists. In this case, the **residual** field indicates the number of bytes that the device failed to complete for this request.

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.
Note: Calling **DK_PASSTHRU** ioctl as a non-root user fails with **EACCES** instead of **EPERM**.

Item **Description**

DKPRES_READKEYS

When the device is successfully opened, the **DKPRES_READKEYS** operation provides a means to read the Persistent Reserve Registration Keys on the device. The *arg* parameter to the **DKPRES_READKEYS** contains the address of the **dk_pres_in** structure. This structure is defined in **/usr/include/sys/scdisk.h**. The user must provide a buffer area and size for the registered keys to be returned. The *returned_length* variable sets the number of bytes returned.

In a shared-access or clustered environment, this operation identifies all registered keys for a particular lun.

Note: For the **DKPRES_READKEYS** operation and following Persistent Reserve related operation, the interpretation of the returned value and scsi status is as follows:

- On successful attempt of the call, a 0 is returned.
- After a call fails, a -1 is returned and the **errno** global variable is set. For a specific description of the **errno** value, refer to **/usr/include/erno.h**. In addition, the SCSI status, along with the Sense Code, ASC and ASCQ, is set to provide further information about why the command failed. Refer to SCSI Specification on the interpretation of the SCSI status failure code.

DKPRES_READRES

When the device is successfully opened, the **DKPRES_READRES** operation provides a means to read the Persistent Reserve Reservation Keys on the device. The *arg* parameter to the **DKPRES_READKEYS** contains the address of the **dk_pres_in** structure. This structure is defined in **/usr/include/sys/scdisk.h**. The user must provide a buffer area and size for the reservation information to be returned. The *returned_length* variable sets the number of bytes returned. In a shared-access or clustered environment, this operation identifies the primary initiator that holds the reservation.

DKPRES_CLEAR

When the device is successfully opened, the **DKPRES_CLEAR** operation provides a means to clear all Persistent Reserve Reservation Keys and Registration Keys on the device. The *arg* parameter to **DKPRES_CLEAR** contains the address of the **dk_pres_clear** structure. This structure is defined in **/usr/include/sys/scdisk.h**.

Attention: Exercise care when issuing the **DKPRES_CLEAR** operation. This operation leaves the device unreserved, which allows a foreign initiator to access the device.

DKPRES_PREEMPT

When the device is successfully opened, the **DKPRES_PREEMPT** operation provides a means to preempt a Persistent Reserve Reservation Key or Registration Key on the device. The *arg* parameter to the **DKPRES_PREEMPT** contains the address of the **dk_pres_preempt** structure. This structure is defined in **/usr/include/sys/scdisk.h**. The user must provide the second party initiator key on the device to be preempted. If the second party initiator holds the reservation to the device, then the initiator that issues the preemption becomes the owner of the reservation. Otherwise, the second party initiator access is revoked.

In order for this operation to succeed, the initiator must be registered with the device first before any preemption can occur. In a shared-access or clustered environment, this operation is used to preempt any operative or inoperative initiator, or any initiator that is not recognized to be part of the shared group.

DKPRES_PREEMPT_ABORT

This operation is the same as the **DKPRES_PREEMPT**, except the device follows the SCSI Primary Command Specification in canceling tasks that belong to the preempted initiator.

DKPRES_REGISTER

When the device is successfully opened, the **DKPRES_REGISTER** operation provides a means to register a Key with the device. The Key is extracted from ODM Customize Attribute and passed to the device driver during configuration. The *arg* parameter to the **DKPRES_REGISTER** contains the address of the **dk_pres_register** structure. This structure is defined in **/usr/include/sys/scdisk.h**.

In a shared-access or clustered environment, this operation attempts a registration with the device, then follows with a read reservation to determine whether the device is reserved. If the device is not reserved, then a reservation is placed with the device.

Item

Description

DK_RWBUFFER

When the device is 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 performs 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 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 must be set according to the required 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 quiesces all other I/O from the initiator by issuing the Write Buffer ioctl until the entire operation completes. Once the Write Buffer ioctl completes, all quiesced I/O are 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 must evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions must be taken.

If a **DK_RWBUFFER** operation fails because a field in the **sc_rwbuffer** structure has an invalid value, the subroutine returns a value of -1 and sets 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. It is useful for applications such as microcode download where the user provides a pointer to the entire microcode image, but, because of size restrictions of the device buffers, 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 is 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 are issued to the device. The number of Write Buffer commands (SCSI CDBs) issued are calculated by dividing the **data_length** by the required **fragment_size**. This value is incremented by 1 if the **data_length** is not an even multiple of **fragment_size**, and the final data transfer is the size of this residual amount. For each Write Buffer command that is issued, the **buffer_offset** is 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** is set to the value provided in the **sc_rwbuffer** struct. For each subsequent Write Buffer command that is issued, the **buffer_ID** is incremented by 1 until all fragments are 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 request is quiesced between ioctl invocations.

Item **Description**

DK_RWBUFFER continued

If **fragment_size** is set to zero, an **errno** of **EINVAL** is returned. If the desire is to send the entire buffer with one SCSI Write buffer command, this field must be set equal to **data_length**. An error of **EINVAL** is also 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 results in an SCSI error at the device, and the Write Buffer ioctl results 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 that is 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 must 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 tracks 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 overrides the **DKPATHFORCE** option and send the I/O down the path that is 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 that is 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 that is 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 architectural model hard disk, CD-ROM, and read/write optical drives have the following hardware requirements:

- SAM hard 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 SAM hard disks and read/write optical drives.
- SAM hard 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 hard disk or read/write optical drive does not support this feature, the system error log might indicate premature drive failure.
- SAM CD-ROM and read/write optical drives must support the 10-byte SCSI read command.
- SAM hard 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 feature, the user can write an application for the drive so that it formats media by using the **DKFORMAT** operation.
- If a SAM CD-ROM drive uses **CD_ROM Data Mode 1**, it must support a block size of 512 bytes per block.
- If a SAM CD-ROM drive uses **CD_ROM data Mode 2 Form 1**, it must support a block size of 2048 bytes per block.
- If a SAM CD-ROM drive uses **CD_ROM data Mode 2 Form 2**, it must support a block size of 2336 bytes per block.
- If a SAM CD-ROM drive uses **CD_DA** mode, it must support a block size of 2352 bytes per block.
- To control volume by 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 subchannel
  - pause resume
  - play audio MSF
  - play audio track index
  - read TOC

Error Conditions

Possible **errno** values for **ioctl**, **open**,**read**, and **write** subroutines when you use the **scsidisk** device driver include:

Item	Description
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 invalid user address.
EFORMAT	Indicates that the target device has unformatted media or media in an incompatible format.
EINPROGRESS	Indicates that a CD-ROM drive has a play-audio operation in progress.

Item **Description**

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 incorrect configuration command is 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 is indicated an unrecoverable hardware error.

EMEDIA Indicates one of the following circumstances:

- The target device is indicated an unrecoverable media error.
- The media was changed.

EMFILE Indicates that 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 that 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 hard disk.

EPERM Indicates that the attempted subroutine requires appropriate authority.

ESTALE Indicates that 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 that an I/O operation exceeded the specified timer value.

EWRPROTECT Indicates one of the following circumstances:

- An **open** operation that requests **read/write** mode was attempted on read-only media.
- A **write** operation was attempted to read-only media.

Reliability and Serviceability Information

SCSI hard disk devices, CD-ROM drives, and read/write optical drives return the following errors:

Item **Description**

ABORTED COMMAND Indicates that the device ended the command.

ADAPTER ERRORS Indicates that the adapter returned an error.

GOOD COMPLETION Indicates that the command completed successfully.

HARDWARE ERROR Indicates an that unrecoverable hardware failure occurred during command execution or during a self-test.

ILLEGAL REQUEST Indicates that an incorrect command or command parameter.

MEDIUM ERROR Indicates that the command ended with an unrecoverable media error condition.

NOT READY Indicates that the logical unit is offline or media is missing.

RECOVERED ERROR Indicates that the command was successful after some recovery was applied.

Item	Description
UNIT ATTENTION	Indicates that the device is reset or the power is turned on.

Error Record Values for Media Errors

The fields that are defined in the error record template for hard disk, CD-ROM, and read/write optical media errors are:

Item	Description
`Comment`	Indicates hard 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 must be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error log entry must 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 must be replaced and tried again • 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 must be replaced and tried again • 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 that are defined in the error record template for hard disk, CD-ROM, and read/write optical hardware errors, as well as hard-aborted command errors are:

Item	Description
`Comment`	Indicates hard 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 must be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error log entry must 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 must 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 that are defined in the error record template for hard disk, CD-ROM, and read/write optical media errors adapter-detected hardware errors are:

Item	Description
`Comment`	Indicates adapter-detected hard 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 must be included when an error report is generated.
`Log`	Equals a value of True, which indicates an error-log entry must 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.

Item **Description**

`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 must 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 that are defined in the error record template for hard disk, CD-ROM, and read/write optical media errors recovered errors are:

Item **Description**

`Comment`
Indicates hard 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 must be included when an error report is generated.

`Log`
Equals a value of True, which indicates an error log entry must 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 must be performed
- 1601, which indicates the removable media must be replaced and tried again

`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 must be replaced and tried again
- 00E1 Perform problem determination procedures

Item	Description
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 that are defined in the error record template for hard disk, CD-ROM, and read/write optical media errors unknown errors are:

Item	Description
Comment	Indicates hard 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 must be included when an error report is generated.
Log	Equals a value of True, which indicates an error log entry must 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 must be replaced and tired again
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) occurs if devices that support paging, logical volumes, or mounted file systems are accessed by 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 must not be used for other purposes.

The special files that are used by the **scsidisk** device driver include the following (listed by type of device):

- Hard disk devices:

Item	Description
/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 hard disks.
/dev/hdisk0, **/dev/hdisk1**,..., **/dev/hdisk***n*	Provide an interface to allow SCSI device drivers block I/O access to SCSI hard disks.

- CD-ROM devices:

Item	Description
/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:

Item	Description
/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 that the drive is accessed as a raw device rather than a block device. Performing raw I/O with a hard 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:

Understanding the scsi_buf Structure

open, openx, or creat

write, writex, writev, or writevx

rhdisk subroutine

scsisesdd SAM Device Driver

Purpose

Supports the **Serial Attached 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.

The **open** subroutine 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 it is not currently opened. If an attempt is made to open a **device** that 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.
SESPASSTHRU	When a 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 through the *arg* parameter, which contains the address of an **sc_passthru** structure (defined in the **/usr/include/sys/scsi.h** file). If the **SESPASSTHRU** 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 must evaluate the returned status bytes to determine why the operation failed and what recovery actions must be taken. The **version** field of the **sc_passthru** structure should be set to the value of **SCSI_VERSION_1**, and SES does not support Variable length CDBs. On completion of the SESPASSTHRU ioctl request, the **residual** field indicates the leftover data that the device did not fully satisfy for this request. Upon successful completion, the **residual** field indicates that the device does not have all the data that was requested or the device has less than the amount of data that was requested. Upon failure, the user needs to check the **status_validity** field to determine if a valid SCSI bus problem exists. In this case, the **residual** field indicates the number bytes that the device failed to complete for this request. 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.

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 an SCSI ID different from the SCSI IDs of 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 is 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 requests that are not valid 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, and so on).

Examples

This is the example code for filling the **sc_passthru** structure for the **SESPASSTHRU** ioctl to issue Standard Inquiry SCSI CDB:

```
struct sc_passthru passthru;
passthru.version = SCSI_VERSION_1;
passthru.timeout_value = 30;
passthru.command_length = 6;
passthru.q_tag_msg = SC_SIMPLE_Q;
passthru.flags = B_READ;
passthru.autosense_length = SENSE_LEN;
passthru.autosense_buffer_ptr = &sense_data[0];    /* Buffer for Auto Sense Data */
passthru.data_length = 0xFF;
passthru.buffer = data;            /* Data buffer address to store inquiry data */
passthru.scsi_cdb[0] = SCSI_INQUIRY;
passthru.scsi_cdb[1] = 0x00;
passthru.scsi_cdb[2] = 00; /* Page Code */
passthru.scsi_cdb[3] = 00;
passthru.scsi_cdb[4] = 0xFF;
passthru.scsi_cdb[5] = 0x00;
```

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.
EEXIST	Device already exists in the device table.
ENOMEM	Memory allocation failed.
EFAULT	An illegal user address was entered.
EINVAL	The data buffer length exceeded the maximum defined in the **devinfo** structure for a **SESPASSTHRU** ioctl operation.
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 was given.
EINVAL	The **variable_cdb_ptr** or **variable_cdb_length** fields are set in the **sc_passthru** struct.
EIO	The target device cannot be located or is not responding.
EIO	The target device has indicated an unrecovered hardware error.
EMFILE	An open operation 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 was not defined.
ENODEV	An attempt was made to close a device that was not defined.
ENXIO	The parameter or device number supplied by the **ioctl** subroutine is not valid, or the device is not configured.
EPERM	The attempted subroutine requires appropriate authority.
ETIMEDOUT	An I/O operation has exceeded the given timer value.

Files

Item	Description
/dev/ses0, /dev/ses1... /dev/sesn	Provides an interface to allow SCSI device drivers access to SCSI enclosure services devices.

Related information:

SAM Subsystem Overview

A Typical Initiator-Mode SAM Driver Transaction Sequence

SAM Adapter Device Driver ioctl Commands

Understanding the Execution of Initiator I/O Requests

sctape FC Device Driver

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.

Purpose

Supports the Fibre Channel Protocol for SCSI (FCP) for sequential access bulk storage medium device driver.

Syntax

```
#include <sys/devinfo.h>
#include <sys/scsi.h>
#include <sys/tape.h>
#include <sys/pcm.h>
#include <sys/mpio.h>
```

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 **open** subroutine applies a reservation policy based on the ODM **reserve_policy** attribute.

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:

Item	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 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 (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.

FCP Options to the openx Subroutine in *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.

The following ioctl operations are only available for MPIO capable FC tape devices:

Item	Description
STPATHIOCMD	The **STPATHIOCMD** command will take as input a pointer argument which points to a single sctape_pathiocmd structure. The **STPATHIOCMD** command will behave exactly like the **STIOCMD** command, except that the input path is used rather than normal path selection performed by the PCM. The **STPATHIOCMD** path is used for the **STIOCMD** command regardless of any path specified by a **STPATHFORCE** ioctl command. A path cannot be unconfigured while it is being forced.
STPATHFORCE	The **STPATHFORCE** command takes as input a *ushort* path ID. The path ID should correspond to one of the path IDs in the **CuPath** ODM. The path ID specifies a path to be used for all subsequent I/O commands, overriding any previous **STPATHFORCE** paths. A zero (0) 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. I/O commands sent in with **STPATHIOCMD** will override the **STPATHFORCE** option and send the I/O down the path specified in the **st_pathiocmd** structure.
STPATHPASSTHRU	The **STPATHPASSTHRU** command takes as input a pointer argument that points to a single **sctape_pathiocmd** structure. The **STPATHPASSTHRU** command will behave exactly like **STIOCMD**, except that the input path is used rather than normal path selection.
STPCMPASSTHRU	The **STPCMPASSTHRU** command takes as input a structure that 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.

Error Conditions

In addition to those errors listed, **ioctl**, **open**, **read**, and **write** subroutines against this device are unsuccessful in the following circumstances:

Item	Description
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 the beginning of the 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 an **open** operation was attempted for an adapter that already has the maximum permissible number of opened 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:

Item	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 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. This condition may be caused by a tape flaw or a dirty head.
NOT READY	Indicates the logical unit is offline.
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.

Medium, hardware, and aborted command errors from the preceding list are to be logged every time they occur. The **ABORTED COMMAND** error might 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:

Item	Description
`Comment`	Equal to tape media error.
`Class`	Equal to H, indicating 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 1332, which indicates a tape operation failure.
`Prob_Causes`	Equals a value of 5003, which indicates tape media.
`User_Causes`	Equals a value of 5100 and 7401, which indicates a cause originating with the tape and defective media, respectively.
`User_Actions`	Equal to 1601 and 0000, which indicates, 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, which indicates tape media.
`Fail_Actions`	Equal to 1601 and 0000, which indicates, 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.

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:

Item	Description
`Comment`	Equal to a tape hardware or aborted command 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 1331, which indicates a tape drive failure.
`Prob_Causes`	Equals a value of 6314, which indicates 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.

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:

Item	Description
`Comment`	Indicates the tape-recovered error threshold has been exceeded.
`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 1331, which indicates a tape drive failure.
`Prob_Causes`	Equal to 6314, which indicates the probable cause is the tape drive.
`User_Causes`	Equal to 5100 and 7401, which indicates that the media is defective and the read/write head is dirty, respectively.
`User_Actions`	Equal to 1601 and 0000, which indicates 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, which indicates the cause is the tape and tape drive, respectively.
`Fail_Actions`	Equals a value of 0000, which indicates 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. 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.

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:

Item	Description
`Comment`	Equal to a tape FC adapter-detected 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 1331, which indicates a tape drive failure.
`Prob_Causes`	Equals values of 3300 and 6314, which indicates an adapter and tape drive failure, respectively.
`User_Causes`	None.
`User_Actions`	Equals a value of 0000, which indicates that problem determination procedures should be performed.
`Inst_Causes`	None.
`Inst_Actions`	None.
`Fail_Causes`	Equals values of 3300 and 6314, which indicates an adapter and tape drive failure, respectively.
`Fail_Actions`	Equals a value of 0000, which indicates problem-determination procedures should be performed.

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.

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.

Item	Description
`Comment`	Indicates that the tape drive needs cleaning.
`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 1332, which indicates a tape operation error.
`Prob_Causes`	Equals a value of 6314, which indicates that the probable cause is the tape drive.
`User_Causes`	Equal to 7401, which indicates a dirty read/write head.
`User_Actions`	Equals a value of 0000, which indicates that problem determination procedures should be performed.
`Inst_Causes`	None.
`Inst_Actions`	None.
`Fail_Causes`	Equals a value of 6314, which indicates that the cause is the tape drive.
`Fail_Actions`	Equals a value of 0000, which indicates problem-determination procedures should be performed.

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.

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:

Item	Description
`Comment`	Equal to a tape unknown error.
`Class`	Equal to all error classes.
`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 0xFE00, which indicates the error description is unknown.
`Prob_Causes`	Equals the following values: • 3300, which indicates a tape drive failure • 5003, which indicates a tape failure • 6314, which indicates an adapter failure

Item	Description
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	Equals a value of 0xFFFF, which indicates the failure causes are unknown.
Fail_Actions	Equals 0000, which indicates 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 *Fibre Channel (FC) Specification* for the applicable device for the format of the particular request-sense information.

Related information:

Special Files Overview

FCP Device Driver

ioctl or ioctlx

read, readx, readv, or readvx

tape 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 *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 **scsi_iocmd** structure. This structure is defined in the **/usr/include/sys/scsi_buf.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.

Error	Description
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.
`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,...,

Item	Description
/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 reference:

"Parallel SCSI Adapter Device Driver" on page 143

Related information:

rhdisk subroutine

ioctl subroutine

A Typical Initiator-Mode SCSI Driver Transaction Sequence

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 timeout 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 timeout 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 timeout 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 timeout as shown under the description of the write entry point. The default **send** command timeout value is a scaled timeout 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 timeout" 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

Item	Description
/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related reference:

"scdisk SCSI Device Driver" on page 151

"tmscsi SCSI Device Driver" on page 227

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related reference:

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

"tmscsi SCSI Device Driver" on page 227

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

Item	Description
/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related reference:
"tape SCSI Device Driver" on page 215
"scdisk SCSI Device Driver" on page 151
"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/dev/tmscsi0, **/dev/tmscsi1,...**, **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
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

Item	Description
/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/dev/tmscsi0, **/dev/tmscsi1**,..., **/dev/tmscsi***n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related reference:

"tape SCSI Device Driver" on page 215

"scdisk SCSI Device Driver" on page 151

"Parallel SCSI Adapter Device Driver" on page 143

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

Item	Description
/dev/tmscsi0,/dev/tmscsi1,..., /dev/tmscsi*n*	Support processor-to-processor communications through the SCSI target-mode device driver.

Related reference:
"tape SCSI Device Driver" on page 215
"scdisk SCSI Device Driver" on page 151
"Parallel SCSI Adapter Device Driver" on page 143

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.

Value	Description
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 reference:

"Parallel SCSI Adapter Device Driver" on page 143

Related information:

tmscsi subroutine

errpt command

NVMe subsystem

Provides device driver support for Non-Volatile Memory Express (NVMe). The device driver supports Peripheral Component Interconnect Express (PCIe) attachment of storage that conforms to the NVMe specification.

The NVMe protocol stack consists of a single device driver that supports interfaces to both the PCIe-attached NVMe controller device and corresponding NVMe storage (`hdisk`) devices.

NVMe storage (hdisk) device driver

Purpose

Supports Peripheral Component Interconnect Express (PCIe)-attached Non-Volatile Memory Express (NVMe) storage devices.

Syntax

```
<#include /usr/include/sys/nvme.h>
<#include /usr/include/sys/devinfo.h>
```

Description

The `/dev/hdiskn` special file provides interfaces to the NVMe storage device driver.

Device-dependent subroutines

The NVMe storage device driver supports the `open`, `close`, `read`, `write`, and `ioctl` subroutines.

ioctl subroutine

Along with the IOCINFO operation, the NVMe storage device driver defines operations for NVMe storage devices.

The IOCINFO operation is defined for all device drivers that use the `ioctl` subroutine, as follows:

The IOCINFO operation returns a `devinfo` structure. The `devinfo` structure is defined in the `/usr/include/sys/devinfo.h` header file. The device type in this structure is `DD_SCDISK`, and the subtype is `DS_PV`. The flags field is used to indicate the values `DF_SSD` and `DF_NVME`. When the **`DF_4B_ALINGED`** flag is preset, the flag indicates that all host data buffer addresses must be aligned to a 4 byte address.

NVMe storage ioctl operations

The following `ioctl` operations are supported for NVMe storage devices:

NVME_PASSTHRU

Provides options to send a **passthru** command to an NVMe storage device. The *arg* parameter for the `NVME_PASSTHRU` operation is the address of an `NVME_PASSTHRU` structure that is defined in the `/usr/include/sys/nvme.h` header file.

Note: You can send an **admin** command only to the adapter device, and an **NVM** command only to the storage (`hdisk`) device. Otherwise, the commands can result in undefined behavior such as data corruption. The opcodes of all the NVMe commands are unique only within a command set and the opcode value can be used for different operations in different command sets.

When an **NVMe passthru** command is issued to the NVMe storage device, a specific `path_id` can be specified. If you want to use specific `path_id` for the **passthru** command, you need to configure both the **`NVME_PASS_PASSTHRU`** flag in the `flags` field and the `path_id` to be used in the `path_id` field.

You can write the `dword_10` to `dword_15` data in endian format that is used by the host such that (leftmost) bit 31 is the most significant bit and (rightmost) bit 0 is the least significant bit according to the NVMe specification. For example, to read 0xAC blocks and to set FUA, `dword_10` in the `passthru` structure is set to 0x400000AB.

You must *byte reverse* any two fields in the data that is transferred by using the **passthru** command. For example, to read the name space size (NSZE) from data that is returned by Identify Namespace, bytes 0 - 7 in the data buffer must be *byte reversed*.

The **passthru** command is unsuccessful if the ioctl subroutine returns -1. A return value of -1 indicates that the driver failed to send the command to the controller or the controller did not respond before the timeout occurred. If the errno flag is set to the EINVAL value, the **resp.status** parameter contains a code that indicates the invalid field.

The **passthru** command is successful if the ioctl subroutine returns 0 and if the **resp.status** parameter contains 0. The **passthru** command runs in parallel with commands that are initiated by another user by running read or write operation.

NVMe controller device driver

Purpose

Supports the Non-Volatile Memory Express (NVMe) controller.

Syntax

```
<#include /usr/include/sys/nvme.h>
<#include /usr/include/sys/devinfo.h>
```

Description

The /dev/nvmen special file provides interfaces to the NVMe controller device driver.

Device-dependent subroutines

The NVMe controller device driver supports the open, close, and ioctl subroutines only. The read and write subroutines are not supported by the NVMe controller special file.

ioctl Subroutine

Along with the IOCINFO operation, the NVMe controller device driver defines operations for NVMe controller devices.

The IOCINFO operation is defined for all device drivers that use the ioctl subroutine as follows:

- The IOCINFO operation returns a devinfo structure. The devinfo structure is defined in the /usr/include/sys/devinfo.h header file. The device type in this structure is DD_BUS, and the subtype is DS_NVME. The flags field is not used and it is set to 0.
- The devinfo structure includes unique data such as version information and the data transfer size that is allowed in the maximum initiator mode. The transfer size is specified in bytes.

NVMe controller ioctl operations

The following ioctl operations are supported for NVMe controller devices:

NVME_PASSTHRU

Provides options to send a **passthru** command to an NVMe controller device. The **arg** parameter for the NVME_PASSTHRU operation is the address of an NVME_PASSTHRU structure that is defined in the /usr/include/sys/nvme.h header file.

Note: You can send an **admin** command only to the adapter device, and an **NVM** command only to the storage (hdisk) device. Otherwise, the commands can result in undefined behavior such as data corruption. The opcodes of all the NVMe commands are unique only within a command set and the opcode value can be used for different operations in different command sets.

When an **NVMe passthru** command is issued to the NVMe controller device, a specific path_id can be specified. If you want to use specific path_id for the **passthru** command, you need to configure both the **NVME_PASS_PASSTHRU** flag in the flags field and the path_id to be used in the path_id field.

You can write the dword_10 to dword_15 data in the endian format that is used by the host such that (leftmost) bit 31 is the most significant bit and (rightmost) bit 0 is the least significant bit according to the NVMe specification. For example, to read 0xAC blocks and to set FUA, dword_10 in the passthru structure is set to 0x400000AB.

You must *byte reverse* any fields in the data that is transferred by using the **passthru** command. For example, to read the name space size (NSZE) from data that is returned by Identify Namespace, bytes 0 - 7 in the data buffer must be *byte reversed*.

The **passthru** command is unsuccessful if the ioctl subroutine returns -1. A return value of -1 indicates that the driver failed to send the command to the controller or the controller did not respond before the timeout occurred. If the errno flag is set to the EINVAL value, the **resp.status** parameter contains a code that indicates the invalid field.

The **passthru** command is successful if the ioctl subroutine returns 0 and if the **resp.status** parameter contains 0. The **passthru** command runs in parallel with commands that are initiated by another user by running read or write operation.

NVME_CNTL
Provides the options to submit a control request to the NVMe controller device driver. The **arg** parameter of the NVME_CNTL operation is the address of an nvme_cntl structure that is defined in the /usr/include/sys/nvme.h header file. The types of control operations that are supported for the NVMe controller device driver are documented in the nvme_cntl structure.

USB Subsystem

The protocol stack of the Universal Serial Bus (USB) device driver for the AIX operating system consists of several drivers that communicate with each other in a layered fashion. These layers of drivers in the USB subsystem work together to support the attachment of a range of USB devices, such as flash drives, removable disk drive (RDX), tape, keyboard, mouse, speakers, and optical devices (for example, CD-ROM, CD-R, CD-RW, DVD-R, DVD-RW, and DVD-RAM).

Extensible Host Controller Adapter Device Driver

Purpose

Supports the Universal Serial Bus (USB) 3.0 Extensible Host Controller Interface (xHCI) specification for adapter device drivers.

Syntax

```
#include <sys/hcdi.h>
#include <sys/usbdi.h>
#include <sys/usb.h>
```

Description

The /dev/usbhc*n* special files provide interfaces that allow access to the USB host controller adapter devices. These files manage the adapter resources so that multiple USB client drivers and the USB system (or the protocol driver) can access the USB devices on the same USB host controller adapter simultaneously.

The AIX operating system supports the USB host controllers with various interface architectures, such as the Open Host Controller Interface (OHCI) and the Enhanced Host Controller Interface (EHCI). The binary interface to the USB 3.0 adapters is called the Extensible Host Controller Interface (xHCI). The AIX operating system currently supports the 0.96 and 1.0 versions of the xHCI specification. The xHCI specification defines a new host controller architecture that replaces the existing OHCI or EHCI specification and also extends to new specifications, for example, USB Version 3.0, or later.

The `/usr/lib/drivers/pci/xhcidd` device driver handles the xHCI adapters and the `/usr/lib/methods/cfgxhci` device driver is the corresponding AIX configuration method.

The `max_slots` Object Data Manager (ODM) attribute for the adapter driver specifies the maximum number of USB devices that are supported by an xHCI adapter. The default value of the attribute is 8. You can modify this value to a maximum value of 32 to support more devices.

Note: If the `max_slots` values is set to 8 and if you connect more than 8 USB devices to the USB adapter, the adapter configures only 8 devices. The configuration for the remaining device fails.

Adapter device driver entry point subroutines

The USB adapter device driver supports only the `open`, `close`, `ioctl`, and `config` entry points. The `read` and `write` entry points are not supported.

open and close subroutines

The `open` subroutine associates the device number, which is specified as a parameter to the `open` system call, with the internal adapter device structure. If the `open` subroutine finds an adapter structure, it verifies that the corresponding adapter device is configured and is not marked inactive. If the `open` subroutine does not find an adapter structure, it returns an error. If the Enhanced Error Handling (EEH) feature is enabled, the `open` subroutine prevents access to the device when an EEH event is being processed.

ioctl subroutine

The xHCI adapter device driver supports the following `ioctl` suboperation:

Operation	**Description**
HCD_REGISTER_HC	Registers the call vectors between the USB system (or the protocol driver) and the host controller driver. After the call vectors are registered, all further communication between the USB system (or protocol driver) and the host controller driver is handled by these vectors.

Summary of error conditions returned by the xHCI adapter device driver

The following Transfer Request Block (TRB) completion status codes are returned by the xHCI during status update if the associated error condition is detected. The TRB status values are specified in the xHCI specification. These completion codes in turn are mapped to the following *USBstatus* values:

TRB status	*USBstatus* **value**	**Description**
XHCI_TRB_STATUS_BAB_DET_ERR	USBD_STALL	Babbling during transaction
XHCI_TRB_STATUS_BW_ERR	USBD_ERROR	Bandwidth is not available for periodic endpoint connection
XHCI_TRB_STATUS_BW_OVERRUN_ERR	USBD_ERROR	Isochronous transfer descriptor (TD) exceeded bandwidth of the endpoints
XHCI_TRB_STATUS_CMDRING_ABORT_ERR	USBD_ERROR	Command abort operation
XHCI_TRB_STATUS_CMDRING_STOP_ERR	USBD_ERROR	Command ring stopped
XHCI_TRB_STATUS_CTXT_STATE_ERR	USBD_ERROR	Invalid context state change command
XHCI_TRB_STATUS_DATA_BUF_ERR	USBD_ERROR	Overrun or underrun
XHCI_TRB_STATUS_EP_NE_ERR	USBD_ERROR	Endpoint is in a disabled state

TRB status	*USBstatus* value	Description
XHCI_TRB_STATUS_EVENT_LOST_ERR	USBD_ERROR	Internal event overrun
XHCI_TRB_STATUS_EVTRING_FULL_ERR	USBD_ERROR	Event ring is full
XHCI_TRB_STATUS_INCOMPAT_DEV_ERR	USBD_ERROR	Incompatible device
XHCI_TRB_STATUS_INVALID	USBD_ERROR	Completion update error
XHCI_TRB_STATUS_INVALID_SID_ERR	USBD_ERROR	Invalid stream ID
XHCI_TRB_STATUS_INVALID_STR_TYP_ERR	USBD_ERROR	Invalid stream of context (Ctxt) type
XHCI_TRB_STATUS_ISOCH_BUF_OVR_ERR	USBD_ERROR	Isochronous buffer overrun
XHCI_TRB_STATUS_MAXEL_LARGE_ERR	USBD_ERROR	Maximum exit latency is too large
XHCI_TRB_STATUS_MIS_SERV_ERR	USBD_ERROR	Isochronous endpoint is not serviced
XHCI_TRB_STATUS_NOPING_RESP_ERR	USBD_ERROR	No ping response within endpoint service interval time (ESIT)
XHCI_TRB_STATUS_NOSLOTS_ERR	USBD_ERROR	Exceeded maximum slots
XHCI_TRB_STATUS_PARAM_ERR	USBD_ERROR	Context parameter is invalid
XHCI_TRB_STATUS_RESOURCE_ERR	USBD_ERROR	No adequate resources
XHCI_TRB_STATUS_RING_OVERRUN_ERR	USBD_ERROR	Ring overrun
XHCI_TRB_STATUS_RING_UNDERRUN_ERR	USBD_ERROR	Ring underrun
XHCI_TRB_STATUS_SEC_BW_ERR	USBD_ERROR	Secondary bandwidth error
XHCI_TRB_STATUS_SHORT_PKT_ERR	USBD_SUCCESSS	The packet size is lesser than the transfer descriptor size in the transfer request.
XHCI_TRB_STATUS_SLOT_DISABLED_ERR	USBD_ERROR	Slot is in a disabled state
XHCI_TRB_STATUS_SPLIT_TR_ERR	USBD_ERROR	Split transaction error
XHCI_TRB_STATUS_STALL_ERR	USBD_STALL	Delay detected on TRB
XHCI_TRB_STATUS_STOP_LEN_ERR	USBD_ERROR	Transfer event length is invalid
XHCI_TRB_STATUS_STOPPED_ERR	USBD_ERROR	Stop endpoint command is received
XHCI_TRB_STATUS_SUCCESS	USBD_SUCCESS	Command success
XHCI_TRB_STATUS_TRB_ERR	USBD_ERROR	TRB parameter error
XHCI_TRB_STATUS_UNDEFINED_ERR	USBD_ERROR	Undefined error condition
XHCI_TRB_STATUS_USB_TSX_ERR	USBD_ERROR	No valid response from the device
XHCI_TRB_STATUS_VF_EVTRING_FULL_ERR	USBD_ERROR	Virtual Function (VF) event ring is full

Call vectors

Whenever the USB configuration method is run, it opens the /dev/usb0 USB system driver (USBD) special file and attempts to register each detected and available USB host controller with the USBD by using a USBD_REGISTER_HC ioctl operation. When the USBD_REGISTER_HC ioctl operation is processed, the USBD opens the host controller driver and requests for the registration of call vectors stored within the host controller driver by using an HCD_REGISTER_HC ioctl operation. After the call vectors are registered with the USBD, all further communication between the USBD and the host controller driver is handled by the call vectors. The summary of call vectors follows:

Call vector	Description
hcdConfigPipes	This call vector is provided by the USBD during the enumeration of USB logical device. This call vector supports the xHCD with USBD, however it does not support EHCI or OHCI drivers. It issues a configure endpoint command to the USB device to make the non-control endpoints on the device operational. This call vector is called by the USBD after the configuration selection is complete on the USB device.
hcdDevAlloc	Detects the attachment of a USB logical device. This call vector is provided by the USBD. This call vector supports the Extensible Host Controller Driver (xHCD) with USBD, however it does not support EHCI or OHCI drivers. It enables the slot, sets the USB device address, and allocates the HCD driver resources to use the USB device. It returns the USB address value to the USBD. After this call, the default control endpoint on the USB device is enabled to query the USB protocol-specific data.
hcdDevFree	Detects the removal of a USB logical device. This call vector is provided by the USBD. This call vector supports the xHCD with USBD, however it does not support EHCI or OHCI drivers. It disables the slot and also deallocates the resources that are allocated by the `hcdDevAlloc` call vector.
hcdGetFrame	Obtains the current frame number from the connected host controller. This call vector is provided by the USBD.
hcdPipeAbort	Cancels the processing of an I/O buffer. The pipe that is specified by the I/O buffer is already halted before the `hcdPipeAbort` call vector is called. This call vector is provided by the USBD.
hcdPipeAddIOB	Increases the maximum number of outstanding I/O buffers. This call vector is provided by the USBD.
hcdPipeClear	Clears, unhalts, and restarts the I/O operations on a specific endpoint. When this call vector is called, the function checks whether the ring is in the halted state.
hcdPipeConnect	Creates a pipe connection to an endpoint on a specific USB device.
hcdPipeDisconnect	Removes the previously established pipe connection with the endpoint on a specific USB device.
hcdPipeHalt	Halts a pipe from the perspective of the host controller. All pending I/O operations remain in a pending state. This call vector is provided by the USBD.
hcdPipeIO	Performs the I/O operations on the USB device. The I/O operations can be of the following transfer types: control, bulk, isochronous, and interrupt.
hcdPipeResetToggle	Resets the data synchronization toggle bit to `DATA0`. This call vector is provided by the USBD.
hcdPipeStatus	Obtains the status of the pipe from the host perspective. This call vector is provided by the USBD.
hcdShutdownComplete	Informs the host controller driver that the `usbdReqHCshutdown` request is completed. This call vector is provided by the USBD.
hcdUnconfigPipes	Detects that a device is removed from the system. This call vector is provided by the USBD. This call vector supports the xHCD with USBD, however it does not support EHCI or OHCI drivers. It issues a configure endpoint command with the `Unconfig` bit set to disable all the non-control endpoints on the USB device and deallocate the resources that are allocated by the `hcdConfigPipes` call vector.
hcdUnregisterHC	Unregisters a host controller from the USBD.
usbdBusMap	Maps the memory for bus mastering. This call vector is provided by the xHCD.
usbdPostIOB	Retires an I/O buffer. This call vector is provided by the adapter driver.
usbdReqHCrestart	This call is provided when an error is detected with the adapter and the recovery of adapter driver from this error requires you to restart the adapter.
usbdReqHCshutdown	This call vector is provided during the removal of host controller.
usbdReqHCunregister	The `CFG TERM` function of the adapter driver requests the USBD to unregister the host controller. This call vector is provided during the removal of the host controller.

Related reference:

"USBD Protocol Driver" on page 276

Related information:

usbhc special file

Required USB Adapter Driver ioctl Commands

Enhanced Host Controller Adapter Device Driver

Purpose

Supports the Enhanced Host Controller Interface (EHCI) specification for adapter device drivers.

Syntax

```
#include <sys/hcdi.h>
#include <sys/usbdi.h>
#include <sys/usb.h>
```

Description

The `/dev/usbhc`*n* special files provide interfaces that allow access to the Universal Serial Bus (USB) host controller adapter devices. These files manage the adapter resources so that multiple USB client drivers and the USB system (or the protocol driver) can access the USB devices on the same USB host controller adapter simultaneously.

In the USB 2.0 design, the USB Implementers Forum (USB-IF) implemented single specification, which is known as EHCI, that supports only high-speed data transfers. EHCI-based adapters are multi-function Peripheral Component Interconnect (PCI) devices that consist of virtual host controller functions that are called companion controllers to support Open Host Controller Interface (OHCI) connectivity to USB 1.0 and 1.1 devices. The Object Data Manager (ODM) `alt_usb_ctrl` attribute of the EHCI adapter provides the location values for the companion OHCI controllers.

Adapter device driver entry point subroutines

The USB adapter device driver supports only the `open`, `close`, `ioctl`, and `config` entry points. The `read` and `write` entry points are not supported.

open and close subroutines

The `open` subroutine associates the device number, which is specified as a parameter to the `open` system call, with the internal adapter device structure. If the `open` subroutine finds an adapter structure, it verifies that the corresponding adapter device is configured and is not marked inactive. If the `open` subroutine does not find an adapter structure, it returns an error. If the Enhanced Error Handling (EEH) feature is enabled, the `open` subroutine does not access the device when an EEH event is being processed.

ioctl subroutine

The EHCI adapter device driver supports the following `ioctl` suboperations:

Operation	Description
HCD_REGISTER_HC	Registers the call vectors between the USB system (or the protocol driver) and the host controller driver. After the call vectors are registered, all further communication between the USB system (or the protocol driver) and the host controller driver is handled by these vectors.
HCD_REQUEST_COMPANIONS	Requests port routing information about the companion OHCI host controllers.

Summary of error conditions returned by the EHCI adapter device driver

Possible values of the *USBstatus* return value for the EHCI adapter device driver are as follows:

Error code	Description
USBD_ABORTED	The associated IRP has ended.
USBD_ABORTING	The associated I/O request packet (IRP) is failing.
USBD_ACTIVE	The logical pipe is in operation and is not halted.
USBD_BADHANDLE	The handle that is passed as parameter through the call vector interface is invalid.
USBD_BANDWIDTH	The logical pipe connection has failed because of bandwidth requirements.
USBD_CONNECT	The logical pipe is already connected.
USBD_DATA	Invalid response from the device.
USBD_DISCONNECT	The device that is associated with the transaction is disconnected or removed.
USBD_ERROR	General error condition.
USBD_HALTED	The logical pipe that is associated with the transaction is halted.
USBD_POWER	The device exceeded power budget.
USBD_SPEED	The port reset operation has failed because of device speed mismatch.
USBD_STALL	The logical pipe that is associated with the transaction is delayed.
USBD_TIMEOUT	The I/O operation has timed out.

Call vectors

Whenever the USB configuration method is run, it opens the `/dev/usb0` USB system driver (USBD) special file, and attempts to register each detected and available USB host controller with the USBD by using a `USBD_REGISTER_HC ioctl` operation. When the `USBD_REGISTER_HC ioctl` operation is processed, the USBD opens the host controller driver and requests for the registration of call vectors that are stored within the host controller driver by using an `HCD_REGISTER_HC ioctl` operation. After the call vectors are registered with the USBD, all further communication between the USBD and the host controller driver is handled by the call vectors. The summary of call vectors follows:

Call vector	Description
hcdGetFrame	Obtains the current frame number from the connected host controller. This call vector is provided by the USBD.
hcdPipeAbort	Cancels the processing of an I/O buffer. The pipe that is specified by the I/O buffer is already halted before the `hcdPipeAbort` call vector is called. This call vector is provided by the USBD.
hcdPipeAddIOB	Increases the maximum number of outstanding I/O buffers. This call vector is provided by the USBD.
hcdPipeClear	Clears, unhalts, and restarts the I/O operations on a specific endpoint. When this call vector is called, the function checks whether the ring is in the halted state.
hcdPipeConnect	Creates a pipe connection to an endpoint on a specific USB device.
hcdPipeDisconnect	Removes the previously established pipe connection with the endpoint on a specific USB device.
hcdPipeHalt	Halts a pipe from the perspective of the host controller. All pending I/O operations remain in the pending state. This call vector is provided by the USBD.

Call vector	Description
hcdPipeIO	Performs the I/O operations on the USB device. The I/O operations can be of the following transfer types: control, bulk, isochronous, and interrupt.
hcdPipeResetToggle	Resets the data synchronization toggle bit to `DATA0`. This call vector is provided by the USBD.
hcdPipeStatus	Obtains the status of the pipe from the perspective of the host controller. This call vector is provided by the USBD.
hcdShutdownComplete	Informs the host controller driver that the `usbdReqHCshutdown` request is completed. This call vector is provided by the USBD.
hcdUnregisterHC	Unregisters a host controller from the USBD.
usbdBusMap	Maps the memory for bus-mastering. This call vector is provided by the Extensible Host Controller Driver (xHCD).
usbdPostIOB	Retires an I/O buffer. This call vector is provided by the adapter driver.
usbdReqHCrestart	This call vector is provided when an error is detected in the adapter and the recovery of adapter driver from this error requires you to restart the adapter.
usbdReqHCshutdown	This call vector is provided during the removal of the host controller.
usbdReqHCunregister	The `CFG TERM` function of the adapter driver requests the USBD to unregister the host controller. This call vector is provided during the removal of the host controller.

Related reference:

"USBD Protocol Driver" on page 276

Related information:

usbhc special file

Required USB Adapter Driver ioctl Commands

open subroutine

HCD_REQUEST_COMPANIONS Adapter Device Driver ioctl Operation

Purpose

Requests port routing information about the Open Host Controller Interface (OHCI) companion controllers.

Description

This `ioctl` command is used by the configuration application to determine information about the OHCI companion controller. This information includes the number of root hub ports, the number of companion controllers, and the number of ports per companion controller.

Return values

The following return values are supported:

Value	Description
0	Successful completion.
DEFAULT	The user has insufficient authority to access the data.
EIO	A permanent I/O error occurred.

Open Host Controller Adapter Device Driver

Purpose

Supports the Open Host Controller Interface (OHCI) specification for adapter device drivers.

Syntax

```
#include <sys/hcdi.h>
#include <sys/usbdi.h>
#include <sys/usb.h>
```

Description

The /dev/usbhc*n* special files provide interfaces that allow access to the Universal Serial Bus (USB) host controller adapter devices. These files manage the adapter resources so that multiple USB client drivers and the USB system (or the protocol driver) can access low and full speed of the USB devices on the same USB host controller adapter simultaneously.

The OHCI adapter supports the USB devices (for example, keyboard and mouse) that operate at USB 1.0 and USB 1.1 speeds.

Adapter device driver entry point subroutines

The USB adapter device driver supports only the open, close, ioctl, and config entry points. The read and write entry points are not supported.

open and close subroutines

The open subroutine associates the device number, which is specified as a parameter to the open system call, with the internal adapter device structure. If the open subroutine finds an adapter structure, it verifies that the corresponding adapter device is configured and is not marked inactive. If the open subroutine does not find an adapter structure, it returns an error. If the Enhanced Error Handling (EEH) feature is enabled, the open subroutine prevents access to the device when an EEH event is being processed.

ioctl subroutine

The OHCI adapter device driver supports the following ioctl suboperation:

Operation	**Description**
HCD_REGISTER_HC	Registers the call vectors between the USB system (or the protocol driver) and the host controller driver. After the call vectors are registered, all further communication between the USB system (or protocol driver) and the host controller driver is handled by these vectors.

Summary of error conditions returned by the OHCI adapter device driver

The following error condition codes for OHCI are translated into a *USBstatus* value to inform the USB protocol driver and the client drivers about the error condition. Possible OHCI error conditions and the corresponding *USBstatus* values follow:

OHCI error condition code	***USBstatus* value**	**Description**
OHCI_CC_BitStuffing	USBD_ERROR	General error condition
OHCI_CC_BufferOverrun	USBD_ERROR	General error condition
OHCI_CC_BufferUnderrun	USBD_ERROR	General error condition
OHCI_CC_CRC	USBD_ERROR	General error condition
OHCI_CC_DataOverrun	USBD_ERROR	General error condition
OHCI_CC_DataToggleMismatch	USBD_ERROR	General error condition
OHCI_CC_DataUnderrun	USBD_ERROR	General error condition
OHCI_CC_DeviceNotResponding	USBD_ERROR	General error condition

OHCI error condition code	*USBstatus* value	Description
OHCI_CC_NotAccessed_0	USBD_ERROR	General error condition
OHCI_CC_NotAccessed_1	USBD_ERROR	General error condition
OHCI_CC_NoError	USBD_SUCCESS	Completion of successful transaction
OHCI_CC_PIDCheckFailure	USBD_ERROR	General error condition
OHCI_CC_STALL	USBD_STALL	The logical pipe is delayed
OHCI_CC_UnexpectedPID	USBD_ERROR	General error condition

Call vectors

Whenever the USB configuration method is run, it opens the `/dev/usb0` USB system driver (USBD) special file and attempts to register each detected and available USB host controller with the USBD by using a `USBD_REGISTER_HC ioctl` operation. When the `USBD_REGISTER_HC ioctl` operation is processed, the USBD opens the host controller driver and requests for the registration of call vectors stored within the host controller driver by using an `HCD_REGISTER_HC ioctl` operation. After the call vectors are registered with the USBD, all further communication between the USBD and the host controller driver is handled by the call vectors. The summary of the call vectors follows:

Call vector	Description
hcdGetFrame	Obtains the current frame number from the connected host controller. This call vector is provided by the USBD.
hcdPipeAbort	Cancels the processing of an I/O buffer. The pipe that is specified by the I/O buffer is already halted before the `hcdPipeAbort` call vector is called. This call vector is provided by the USBD.
hcdPipeAddIOB	Increases the maximum number of outstanding I/O buffers. This call vector is provided by the USBD.
hcdPipeClear	Clears, unhalts, and restarts the I/O operations on a specific endpoint. When this call vector is called, the function checks whether the ring is in halted state. This call vector is provided by the USBD.
hcdPipeConnect	Creates a pipe connection to an endpoint on a specific USB device.
hcdPipeDisconnect	Removes the previously established pipe connection with the endpoint on a specific USB device.
hcdPipeHalt	Halts a pipe from the perspective of the host controller. All pending I/O operations remain in a pending state. This call vector is provided by the USBD.
hcdPipeIO	Performs I/O operations on the USB device. The I/O operation can be of the following transfer types: control, bulk, isochronous, and interrupt.
hcdPipeResetToggle	Resets the data synchronization toggle bit to `DATA0`. This call vector is provided by the USBD.
hcdPipeStatus	Obtains the status of the pipe from the host perspective. This call vector is provided by the USBD.
hcdShutdownComplete	Informs the host controller driver that the `usbdReqHCshutdown` request is completed. This call vector is provided by the USBD.
hcdUnregisterHC	Unregisters a host controller from the USBD.
usbdBusMap	Maps memory for bus mastering by the host controller. This call vector is provided by the Extensible Host Controller Driver (xHCD).
usbdPostIOB	Retires an I/O buffer. This call vector is provided by the adapter driver.
usbdReqHCrestart	This call is provided when an error is detected with the adapter and the recovery of adapter driver from this error requires you to restart the adapter.
usbdReqHCshutdown	This call vector is provided during the removal of host controller.
usbdReqHCunregister	The `CFG TERM` function of the adapter driver requests the USBD to unregister the host controller. This call vector is provided during the removal of the host controller.

Related reference:

"USBD Protocol Driver" on page 276

Related information:

usbhc special file

Required USB Adapter Driver ioctl Commands

ioctl subroutine

HCD_REGISTER_HC Adapter Device Driver ioctl Operation

Purpose

Registers the host controller with the Universal Serial Bus (USB) protocol driver.

Description

This `ioctl` command is issued by the USB system driver (USBD) during the registration of host controller with the USBD. During the processing of this `ioctl` operation, the call vectors of the adapter device driver are registered with USBD. After the call vectors are registered with the USBD, all further communication between the USBD and the Host Controller Driver (HCD) is handled by these call vectors.

Return values

The following return values are supported:

Value	Description
0	Successful completion.
DEFAULT	Incorrect size of the call vector or incorrect version of the call vector data structure.
EBUSY	Adapter hardware is inaccessible.
EINVAL	Host controller is already registered with the USBD.

USB Audio Device Driver

Purpose

Supports the Universal Serial Bus (USB) audio devices.

Syntax

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

Description

The USB audio device driver supports isochronous USB devices such as USB audio speakers. Each USB audio device is represented as the following interfaces: audio control and audio streaming. Although these interfaces are associated with the single device, the interfaces are treated as separate devices virtually. The `/dev/paud0` special file is created for audio control interface and the `/dev/paudas0` special file is created for audio streaming interface.

The audio control interface is used to access the internal functions of an audio device. Any request to change the audio controls within the audio function's units or terminals is directed to the audio control interface of the function.

The audio streaming interface can be configured to operate in mono or stereo mode. The number of input channel data streams varies based on the selected mode. Audio streaming interface must have isochronous endpoint. This interface can have alternative settings that can be used to change some characteristics of the endpoint.

Note: You must use external or third-party audio software to stream and play audio files on the supported USB audio devices.

The following table lists the `ioctl` operations:

ioctl operation	Description
AUDIO_INIT	The driver searches the interfaces and alternative settings to determine the setting that can support the requested sample rate, bits per sample, mode, and channels.
AUDIO_STATUS	The driver returns information about its internal data structures.
AUDIO_CONTROL	The driver handles requests to change the audio properties, for example, start, stop, and pause.
AUDIO_BUFFER	The driver calculates and returns the values that are based on the information about its data structures, the amount of data in buffers, the amount of data in requests, the time delay that is specified in the `bDelay` field, and general class-specific interface descriptor.
AUDIO_WAIT	The driver waits until the requests for all remaining playback data are complete. If the `bDelay` field is specified, it waits for that amount of time. This operation must be called just before the `AUDIO_STOP` operation to avoid interruption in the last remaining samples in the playback buffer.
AUDIO_SET_CHANNELS	The driver updates its copies of the record and playback settings in the driver's internal data structures. • If a record path is active and one of the record settings is changed, the driver sends requests to the USB audio device to change the settings in the units. • If the playback path is active and one of the playback settings is changed, the driver sends requests to the USB audio device to change the settings in the units for the playback and playback rider paths. For the playback path, the master settings volume must be included in the calculations before you set the playback path volume.
AUDIO_GET_CHANNELS	The driver returns information that is based on the four input and one output device models.
AUDIO_CHANNEL_STATUS	The driver returns information that is stored in its internal data structures.
AUDIO_SET_GAIN	The driver updates its copy of the settings in its internal data structures. If a record path is active, the driver sends requests to the USB audio device to change the settings in the units.
AUDIO_MODIFY_LIMITS	The driver updates the values in its internal data structures. If a `select()` call is pending and one of the conditions to unblock the `select()` call is met, the `select()` call is unblocked.
AUDIO_MASTER_VOLUME	The driver saves the new master volume value. It calculates the new unit volume value that is based on the new master volume value and the playback volume value. If playback is active, the driver sends requests to the USB audio device to change the settings in the units for the playback and playback rider paths.

USB Keyboard Client Device Driver

Purpose

Supports the Universal Serial Bus (USB) keyboard devices.

Syntax

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

Description

The keyboard client consists of a back end that interfaces with the USB system driver (USBD) and a front end that interfaces with the AIX applications such as the low function terminal (LFT) and X server applications. The USB keyboard client driver has no knowledge of the underlying USB adapter hardware. Instead, the client driver sends control requests to the USB keyboard through the USBD and receives input events through the USBD. The keyboard client driver identifies itself as a generic keyboard driver by setting the devid field in its Object Data Manager (ODM) predefined data to 030101. The parent device of the keyboard client is the pseudo device, usb0. The keyboard client does not have any child. Each keyboard device that is connected to the AIX system is represented as /dev/kbd0, /dev/kbd1, and so on.

The USB keyboard client driver supports the attachment of multiple USB keyboard devices. Each device is enumerated in the ODM and is marked as available. The client driver treats all keyboards as a single logical device. Light-emitting diode (LED) settings are sent to all keyboards and input events from all keyboards are sent to a single input ring. State tracking by the client driver ensures that a key does not generate consecutive break events and that typematic delay and repeat are handled appropriately.

Special files (for example, /dev/kbd0, /dev/kbd1, and so on) are created for each USB keyboard device. If there is at least one available USB keyboard device, an application (typically the LFT or X server application) can open any one of the USB keyboard special files.

Special treatment for the keyboard is provided by the USB system device driver configuration method because of the strict configuration and ordering rules of the graphics subsystem. When both the USB host controller and the graphics adapter are in an available state and no existing keyboard is present, the USBD ensures that at least one USB keyboard instance is defined. The USB keyboard client driver uses the USBD_OPEN_DEVICE_EXT ioctl operation to open the device that generates a valid handle even when no USB keyboard is attached to the system. The EAGAIN value is returned by the ioctl operation if there is no keyboard device and the client driver treats the device as disconnected. When you plug in a USB keyboard, a reconnect call back is made to the keyboard client by the USBD and the device is initialized allowing input events to flow to the LFT and X server applications.

Device-dependent subroutines

The USB adapter device driver supports only the open, close, ioctl, and config subroutines.

open and close subroutines

The open subroutine is used to create a channel between the caller and the keyboard client driver. The keyboard special file supports two such channels. The open subroutine call is processed normally except that the **OFlag** and **Mode** parameters are ignored. The keyboard supports an fp_open request from a kernel process. The keyboard client driver is multiplexed for an orderly change of control between the LFT and the X server applications. The most recently opened keyboard channel is the active channel to which the input events are sent. Only one channel can be open in the kernel mode at a time. The USB keyboard client supports the attachment of multiple USB keyboard devices. Thus, one or more special files can be defined. If there is at least one available USB keyboard device, an application (typically the LFT or the X server) can open any one special file of the USB keyboard because the keyboard client driver ignores the minor number specification. However, only two channels can be defined regardless of the number of available USB keyboard devices.

The close subroutine call is used to end a channel.

Read and write operations

The keyboard client driver does not support read and write operations. A read or write operation to the special file of the driver behaves as if a read operation or a write operation was made to the /dev/null file.

ioctl subroutine

The keyboard device driver supports the following **ioctl** suboperations:

Operation	Description
IOCINFO	Returns a devinfo structure, which is defined in the sys/devinfo.h header file, that describes the device. The first field of the structure (devtype) is set to the DD_INPUT value; the remaining structure is set to zero.
KSQUERYID	Queries keyboard device identifier.
KSQUERYSV	Queries keyboard service vector.
KSREGRING	Registers input ring.
KSRFLUSH	Flushes input ring.
KSLED	Sets or resets keyboard LEDs.
KSVOLUME	Sets alarm volume.
KSALARM	Sounds alarm.
KSTRATE	Sets typematic rate.
KSTDELAY	Sets typematic delay.
KSKAP	Enables or disables keep-alive poll.

Related information:

kbd special file

USB Mass Storage Client Device Driver

Purpose

Supports the Universal Serial Bus (USB) protocol for mass storage and bulk type hard disk, Removable Disk Drive (RDX), flash drives, CD-ROM, DVD-RAM, Blu-ray read-only, and read/write optical memory devices.

Syntax

```
#include <sys/devinfo.h>
#include <sys/scsi.h>
#include <sys/scdisk.h>
#include <sys/ide.h>
#include <sys/usb.h>
#include <sys/usbdi.h>
#include <sys/mstor.h>
```

Description

Typical USB hard disk, RDX, flash drives, CD-ROM, DVD-RAM, Blu-ray read-only, and read/write optical drive operations are implemented by using the open, close, read, write, and ioctl subroutines.

Device-dependent subroutines

The USB mass storage device driver supports only the open, close, ioctl, and config subroutines.

open and close subroutines

The openx subroutine is primarily used by the diagnostic commands and utilities. Appropriate authority is required to run the subroutine. If you run the open subroutine without the required authority, the subroutine returns a value of -1 and sets the *errno* global variable to a value of EPERM.

The *ext* parameter that is specified in the *openx* subroutine selects the operation to be used for the target device. The /usr/include/sys/usb.h file defines the possible values for the *ext* parameter.

The *ext* parameter can contain any logical combination of the following flag values:

Item	Description
SC_DIAGNOSTIC	Places the selected device in the Diagnostic mode. This mode is singularly entrant, which means that only one process at a time can open the device at a time. When a device is in the Diagnostic mode, the USB devices are initialized during the open or close operations, and error logging is disabled. In the 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 the Diagnostic mode only if the target device is not currently opened. If you open a device in the Diagnostic mode when 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_SINGLE	Places the selected device in the Exclusive Access mode. Only one process can open a device in the Exclusive Access mode at a time. A device can be opened in the Exclusive Access mode only if the device is not currently open. If you open a device in the 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.

readx and writex subroutines

The readx and writex subroutines are not supported on USB devices. Even if they are called, the *ext* parameter is not processed.

ioctl subroutine

The ioctl subroutine operations that are used for the usbcd device driver are specific to the following categories of USB devices:

- Common ioctl operations for all USB devices
- USB hard disk, flash drive, and RDX devices
- USB CD-ROM and read/write optical devices

Common ioctl operations supported for all USB devices

The following ioctl operations are available for hard disk, flash drive, RDX, CD-ROM, and read/write optical devices:

Operation	Description
DKIORDSE	Issues a read command to the device and obtains the target-device sense data when an error occurs. If the DKIORDSE operation returns a value of -1 and if the status_validity field is set to the SC_SCSI_ERROR value, 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 of the DKIORDSE operation contains the address of a 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 you transfer more than the maximum limit, the subroutine returns a value of -1 and sets the *errno* global variable to a value of EINVAL. **Note:** The CDIORDSE operation can be substituted for the DKIORDSE operation when the read command is issued to obtain sense data from a CD-ROM device. The DKIORDSE operation is the recommended operation.

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>DKIOCMD</td><td>

When the device is successfully opened in the Normal or Diagnostic mode, the DKIOCMD operation can issue any Small Computer System Interface (SCSI) command to the specified device. The device driver does not log any error recovery or failures of this operation.

The SCSI status byte and the adapter status bytes are returned through the **arg** parameter that contains the address of a sc_iocmd structure, which is 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 must evaluate the returned status bytes to determine the cause of operation failure and the recovery actions.

The devinfo structure defines the maximum transfer size for the command. If you transfer more than the maximum value, the subroutine returns a value of -1 and sets the *errno* global variable to a value of EINVAL.

</td></tr>
<tr><td>DKIOCMD (continued)</td><td>

The following example code issues the DKIOCMD ioctl operation to the usbms0 device to get the SCSI standard inquiry data:

```
    char sense_data[255];
    char *data_buffer=NULL;
    struct sc_iocmd sciocmd;
....

    fd = open("/dev/usbms0", O_RDWR);
    if (fd == -1){
        printf("\niocmd: Open FAIL\n");
        exit(-1);
    }

    memset(&sciocmd, '\0', sizeof(struct scsi_iocmd));
    sciocmd.version = SCSI_VERSION_1;
    sciocmd.timeout_value = 30;
    sciocmd.command_length = 6;
    sciocmd.flags = B_READ;
    sciocmd.autosense_length = 255;
    sciocmd.autosense_buffer_ptr = &sense_data[0];

    sciocmd.data_length = 0xFF;
    sciocmd.buffer = inq_data;
    sciocmd.scsi_cdb[0] = SCSI_INQUIRY;
    sciocmd.scsi_cdb[1] = 0x00; /* Standard Inquiry*/
    sciocmd.scsi_cdb[2] = 0x00;
    sciocmd.scsi_cdb[3] = 0x00;
    sciocmd.scsi_cdb[4] = 0xFF;
    sciocmd.scsi_cdb[5] = 0x00;

    if ((rc=ioctl(fd, DKIOCMD, &sciocmd)) != 0){
        printf("iocmd: Ioctl FAIL errno %d\n",errno);
        printf("status_validity: %x, scsi_status: %x, adapter_status:%x\n",
               sciocmd.status_validity, sciocmd.scsi_bus_status,
               sciocmd.adapter_status);
        hexdump(sense_data, (long)20);
        close(fd);
        exit(-1);
    } else {
        printf("cdiocmd : Ioctl PASS\n");
        if (cmd = SCSI_INQUIRY)
            hexdump(inq_data,0x20);
    }

    close(fd);
```

</td></tr>
</table>

Operation	Description
DKIOLCMD	When the device is successfully opened in the `Normal` or `Diagnostic` mode, the `DKIOLCMD` operation can issue any SCSI command to the specified device. The device driver does not log any error recovery failures of this operation. This `ioctl` operation is similar to the `DKIOCMD16` operation that is used to issue 16-byte SCSI commands to the USB mass storage device. The SCSI status byte and the adapter status bytes are returned through the **`arg`** parameter that contains the address of a `sc_iocmd16cdb` structure. This structure is defined in the `/usr/include/sys/scsi.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 must evaluate the returned status bytes to determine the cause of operation failure and the recovery actions. On completion of the `DKIOLCMD` `ioctl` request, the residual field indicates the leftover data that the device did not fully satisfy for this request. On a successful completion, the residual field indicates that the device does not have all of the data that is requested or the device has less amount of data than requested. On a request failure, you must check the `status_validity` field to determine whether a valid SCSI bus problem exists. In this case, the residual field indicates the number of bytes that the device failed to complete for this request. The `devinfo` structure defines the maximum transfer size for the command. If you transfer more than the maximum value, the subroutine returns a value of -1 and sets the *errno* global variable to a value of `EINVAL`.
DKIOLCMD (continued)	The following example code issues the `DKIOLCMD` ioctl operation to the `usbms0` device to get the SCSI standard inquiry data:

```
    char sense_data[255];
    char *data_buffer=NULL;
    struct sc_iocmd16cdb sciocmd;
....

    fd = open("/dev/usbms0", O_RDWR);
    if (fd == -1){
        printf("\niocmd: Open FAIL\n");
        exit(-1);
    }

    memset(&sciocmd, '\0', sizeof(struct scsi_iocmd));
    sciocmd.version = SCSI_VERSION_1;
    sciocmd.timeout_value = 30;
    sciocmd.command_length = 6;
    sciocmd.flags = B_READ;
    sciocmd.autosense_length = 255;
    sciocmd.autosense_buffer_ptr = &sense_data[0];

    sciocmd.data_length = 0xFF;
    sciocmd.buffer = inq_data;
    sciocmd.scsi_cdb[0] = SCSI_INQUIRY;
    sciocmd.scsi_cdb[1] = 0x00; /* Standard Inquiry*/
    sciocmd.scsi_cdb[2] = 0x00;
    sciocmd.scsi_cdb[3] = 0x00;
    sciocmd.scsi_cdb[4] = 0xFF;
    sciocmd.scsi_cdb[5] = 0x00;

    if ((rc=ioctl(fd, DKIOCMD, &sciocmd)) != 0){
        printf("iocmd: Ioctl FAIL errno %d\n",errno);
        printf("status_validity: %x, scsi_status: %x, adapter_status:%x\n",
               sciocmd.status_validity, sciocmd.scsi_bus_status,
               sciocmd.adapter_status);
        hexdump(sense_data, (long)20);
        close(fd);
        exit(-1);
    } else {
        printf("cdiocmd : Ioctl PASS\n");
        if (cmd = SCSI_INQUIRY)
            hexdump(inq_data,0x20);
    }

    close(fd);
```

Operation	**Description**
DK_PASSTHRU	When the device is successfully opened, the DK_PASSTHRU operation can issue any SCSI command to the specified device. The device driver performs limited error recovery if this operation fails. The DK_PASSTHRU operation differs from the DKIOCMD operation such that it does not require an openx command with the ext argument of the SC_DIAGNOSTIC field. Because of this, the 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 that contains the address of a sc_passthru structure. This structure is 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. In this case, the caller must evaluate the returned status bytes to determine the cause of operation failure and the recovery actions. If a DK_PASSTHRU operation fails because a field in the sc_passthru structure has an invalid value, the subroutine returns a value of -1 and set the *errno* global variable to EINVAL. The einval_arg field is 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 that no additional information about the failure is available. The version field of the sc_passthru structure can be set to the value of SCSI_VERSION_2 and you can specify the following fields: • The variable_cdb_ptr field is a pointer to a buffer that contains the *cdb* variable. • The variable_cdb_length field determines the length of the *cdb* variable to which the variable_cdb_ptr field points. On completion of the DK_PASSTHRU request, the residual field indicates the leftover data that the device did not fully satisfy for this request. On a successful completion, the residual field indicates that the device does not have all of the data that is requested or the device has less amount of data than requested. On a request failure, you must check the status_validity field to determine if a valid SCSI bus problem exists. In this case, the residual field indicates the number of bytes that the device failed to complete for this request. 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. These values are defined in the /usr/include/sys/scsi.h file. **Note:** If you call the DK_PASSTHRU operation as a non-root user, the operation fails with the EACCES error value instead of the EPERM value.

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>DK_PASSTHRU (continued)</td><td>

The following example code issues the DK_PASSTHRU ioctl operation to the usbms0 device to get the SCSI standard inquiry data:

```
    char sense_data[255];
    char *data_buffer=NULL;
    struct sc_passthru sciocmd;
....

    fd = open("/dev/usbms0", O_RDWR);
    if (fd == -1){
        printf("\npassthru: Open FAIL\n");
        exit(-1);
    }

    memset(&sciocmd, '\0', sizeof(struct sc_passthru));
    sciocmd.version = SCSI_VERSION_1;
    sciocmd.timeout_value = 30;
    sciocmd.command_length = 6;
    sciocmd.autosense_length = 255;
    sciocmd.autosense_buffer_ptr = &sense_data[0];

    sciocmd.data_length = 0xFF;
    sciocmd.buffer = inq_data;

    sciocmd.flags = B_READ;

    sciocmd.scsi_cdb[0] = SCSI_INQUIRY;
    sciocmd.scsi_cdb[1] = 0x00; /* Standard Inquiry*/
    sciocmd.scsi_cdb[2] = 0x00;
    sciocmd.scsi_cdb[3] = 0x00;
    sciocmd.scsi_cdb[4] = 0xFF;
    sciocmd.scsi_cdb[5] = 0x00;

    if ((rc=ioctl(fd, DK_PASSTHRU, &sciocmd)) != 0){
        if (sciocmd.adap_set_flags & SC_AUTOSENSE_DATA_VALID) {
            /* look at sense data */
        } /* end SC_AUTOSENSE_DATA_VALID */

        printf("passthru: Ioctl FAIL errno %d\n",errno);
        printf("status_validity: %x, scsi_status: %x, adapter_status:%x\n",
               sciocmd.status_validity, sciocmd.scsi_bus_status,
               sciocmd.adapter_status);
        printf("Residual: %x\n", sciocmd.residual);
        exit(-1);
    } else {
        printf("passthru: Ioctl PASS\n");
        printf("status_validity: %x, scsi_status: %x, adapter_status:%x\n",
               sciocmd.status_validity, sciocmd.scsi_bus_status,
               sciocmd.adapter_status);
        printf("Residual: %x\n", sciocmd.residual);
        /* inq_data buffer has valid Standard Inquiry data */
    }
```

</td></tr>
</table>

ioctl operations for USB hard disk, flash drive, and RDX devices

The following ioctl operations are available for USB hard disk, flash drive, and RDX devices only:

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>IOCINFO</td><td>

Returns the devinfo structure that is defined in the /usr/include/sys/devinfo.h file. The IOCINFO operation is the only operation that is defined for all device drivers that use the ioctl subroutine. The following values are returned:

```
devinfo.devtype = DD_SCDISK;
devinfo.flags =(uchar)DF_RAND;
devinfo.devsubtype = 0x00;
devinfo.un.scdk.max_request = Maximum_transfer_supported_by_usbcd_driver;
devinfo.un.scdk.numblks = Largest_LBA_supported_by_device+1;
devinfo.un.scdk.blksize = Block_size_set_for_the_USB_Disk/Flash/RDX_Device;
```

</td></tr>
</table>

Operation	Description
DKPMR	Issues an SCSI prevent media removal (PMR) command when the device is successfully opened. This command prevents media from being ejected until the device is closed, powered off and restarted, or until a DKAMR operation is issued. The arg parameter for the DKAMR operation is null. If the DKAMR operation is successful, the subroutine returns a value of 0. If the device is an SCSI hard disk, the DKAMR operation fails, the subroutine returns a value of -1, and sets the *errno* global variable to a value of EINVAL. If the DKAMR operation fails for any other reason, the subroutine returns a value of -1 and sets the *errno* global variable to a value of EIO. **Note:** This function is provided to support the USB RDX devices that support ejecting the media cartridges.
DKAMR	Issues an allow media removal (AMR) command when the device is successfully opened. The media can then be ejected by using either the driver's eject button or the DKEJECT operation. The arg parameter for this ioctl operation is null. If the DKAMR operation is successful, the subroutine returns a value of 0. If the device is an SCSI hard disk, the DKAMR operation fails. In addition, the subroutine returns a value of -1 and sets the *errno* global variable to a value of EINVAL. For any other cause of failure of this operation, the subroutine returns a value of -1, and sets the *errno* global variable to a value of EIO. **Note:** This function is provided to support the USB RDX devices that support ejecting the media cartridges.

ioctl operations for CD-ROM and read/write optical devices

The following ioctl operations are available for CD-ROM and read/write optical devices:

Operation	Description

IOCINFO	Returns the devinfo structure that is defined in the /usr/include/sys/devinfo.h file. The IOCINFO operation is the only operation that is defined for all device drivers that use the ioctl subroutine. The following values are returned:
	devinfo.devtype = DD_CDROM;
	devinfo.flags = (uchar)DF_RAND;
	devinfo.devsubtype = 0x00;
	devinfo.un.idecd.numblks =
	  Largest logical block addressing (LBA) supported by device + 1;
	devinfo.un.idecd.blksize = *Block size set for the USB Disk, flash, or RDX device*;

IDEPASSTHRU	Issues an AT Attachment Packet Interface (ATAPI) command to the specified device when the device is successfully opened. The IDEPASSTHRU operation does not require an **openx** command with the ext argument of the SC_DIAGNOSTIC value. Therefore, an IDEPASSTHRU operation can be issued to devices that are in use by other operations. The AT Attachment (ATA) status bytes and the ATA error bytes are returned through the **arg** parameter. This parameter contains the address of an ide_ata_passthru structure that is defined in the /usr/include/sys/ide.h file. If the IDEPASSTHRU operation fails, the subroutine returns a value of -1 and sets the *errno* global variable to a nonzero value. In this case, the caller evaluates the returned status bytes to determine the cause of operation failure and the recovery actions. If the IDEPASSTHRU operation fails, the device driver performs limited error recovery. If this operation fails because a field in the ide_ata_passthru structure has an invalid value, the subroutine returns a value of -1 and sets the *errno* global variable to EINVAL. On successful completion of the IDEPASSTHRU request, the residual field indicates that the device does not have all of the data that is requested, or the device has less than the amount of data that is requested. If the IDEPASSTHRU request fails, the residual field indicates the number bytes that the device failed to complete for this request.

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>IDEPASSTHRU (continued)</td><td>The following example code issues an SCSI inquiry command that uses the IDEPASSTHRU operation:
<pre><code>   struct ide_atapi_passthru atapicmd;
    char inq_buffer[255];
    uchar sense_data[255];

    /* set up the arg parameter block */
    memset(&atapicmd, '\0', sizeof(struct ide_atapi_passthru));
    memset(sense_data, '\0', 255);

    atapicmd.ide_device = 0;
    atapicmd.flags = IDE_PASSTHRU_READ;
    atapicmd.timeout_value   = 30;
    atapicmd.rsv0   = IDE_PASSTHRU_VERSION_01;
    atapicmd.rsv1   = 0;
    atapicmd.atapi_cmd.length = 12;
    atapicmd.atapi_cmd.resvd = 0;
    atapicmd.atapi_cmd.resvd1 = 0;
    atapicmd.atapi_cmd.resvd2 = 0;

    atapicmd.data_ptr = inq_buffer;
    atapicmd.buffsize = 0xFF;

    atapicmd.atapi_cmd.packet.opcode = SCSI_INQUIRY;
    atapicmd.atapi_cmd.packet.byte[0] = (0x00 | vpd) ; /*Standard Inquiry */
    atapicmd.atapi_cmd.packet.byte[1] = page_code; /*Page Code—Valid if vpd=1 */
    atapicmd.atapi_cmd.packet.byte[2] = 0x00;
    atapicmd.atapi_cmd.packet.byte[3] = 0xFF;
    atapicmd.atapi_cmd.packet.byte[4] = 0x00;

    atapicmd.sense_data   = sense_data;
    atapicmd.sense_data_length = 255;

    fd = openx("/dev/cd0", O_RDWR, NULL, SC_DIAGNOSTIC);
    if (fd == -1) {
         printf("IDEPASSTHRU: Openx failed with errno %x \n", errno);
         exit(-1);
    }
   if ((rc = ioctl(fd, IDEPASSTHRU, &atapicmd) != 0)) {
         printf("IDEPASSTHRU: IOCTL Failed");
         printf("errno %d\n",errno);
         printf("ata_status: %x, ata_error:%x\n",
                atapicmd.ata_status, atapicmd.ata_error);
         close(fd);
         exit(-1);
     } else {
         printf("IDEPASSTHRU : Ioctl PASS\n");
         printf("ata_status: %x, ata_error: %x\n",
                atapicmd.ata_status, atapicmd.ata_error);
     }
     close(fd);</code></pre></td></tr>
<tr><td>DKPMR</td><td>Issues a Small Computer System Interface (SCSI) prevent media removal command when the device is successfully opened. This command prevents media from ejecting until the device is closed, powered off and then powered on, or until a DKAMR operation is issued. The <b>arg</b> parameter for the DKPMR operation is null. If the DKPMR operation is successful, the subroutine returns a value of 0. If the device is an SCSI hard disk, the DKPMR operation fails, the subroutine returns a value of -1, and sets the <i>errno</i> global variable to a value of EINVAL. If the DKPMR operation fails because of any other reason, the subroutine returns a value of -1 and sets the <i>errno</i> global variable to a value of EIO.</td></tr>
<tr><td>DKAMR</td><td>Issues an allow media removal command when the device is successfully opened. The media can be ejected by using either the drives eject button or the DKEJECT operation. The <b>arg</b> parameter for this operation is null. If the DKAMR operation is successful, the subroutine returns a value of 0. If the device is an SCSI hard disk, the DKAMR operation fails, and the subroutine returns a value of -1 and sets the <i>errno</i> global variable to a value of EINVAL. For any other cause of operation failure, the subroutine returns a value of -1 and sets the <i>errno</i> global variable to a value of EIO.</td></tr>
</table>

Operation	Description
DKEJECT	Issues an eject media command to the drive when the device is 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 an SCSI hard disk, the DKEJECT operation fails, the subroutine returns a value of -1, and sets the *errno* global variable to a value of EINVAL. For any other cause of operation failure, the subroutine returns a value of -1 and sets the *errno* variable to a value of EIO.
DKAUDIO	Issues a play audio command to the specified device and controls the volume on the device's output ports. Play audio commands can 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 that is defined in the /usr/include/sys/scdisk.h file. Exclusive access mode is required. If the 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 must evaluate the returned status bytes to determine the cause of operation failure and recovery actions.
DK_CD_MODE	Issues one of the following commands: **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 is successfully opened. **CD_CHG_MODE** Changes the CD-ROM data mode to the mode that is specified in the cd_mode_form field of the mode_form_op structure when the device is successfully opened in the exclusive access mode. If a CD-ROM is not configured for different data modes by using the mode-select density codes, and if you change the CD-ROM data mode by setting the action field of the change_mode_form structure to the CD_CHG_MODE command, 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 also results in a return value of -1 and the *errno* global variable is set to EINVAL. If the DK_CD_MODE operation for the CD_CHG_MODE command 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 cause of operation failure, the subroutine returns a value of -1 and sets the *errno* global variable to a value of EIO.

Device hardware requirements

USB hard disk, flash drive, RDX, CD-ROM, and read/write optical drives have the following hardware requirements:

- These 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 sequential access memory (SAM) hard disks and read/write optical drives.
- USB hard disks, flash drives, RDX, and read/write optical drives must report the hardware retry count in bytes of the request sense data for recovered errors. If the USB hard disk or read/write optical drive does not support this feature, the system error log might indicate premature drive failure.
- USB CD-ROM and read/write optical drives must support the 10-byte SCSI read command.
- USB hard disks, flash drives, RDX, and read/write optical drives must support the SCSI write and verify command and the 6-byte SCSI write command.
- The read/write optical drive must set the format options valid (FOV) bit to 0 for the defect list header of the SCSI format unit command to use the format command operation. If the drive does not support this feature, you can write an application for the drive so that it formats the media by using the DKFORMAT operation.
- If a USB CD-ROM drive uses CD_ROM Data Mode 1 format, it must support a block size of 512 bytes per block.

- If a USB CD-ROM drive uses CD_ROM data Mode 2 Form 1 format, it must support a block size of 2048 bytes per block.
- If a USB CD-ROM drive uses CD_ROM data Mode 2 Form 2 format, it must support a block size of 2336 bytes per block.
- If a USB CD-ROM drive uses CD_DA mode, it must support a block size of 2352 bytes per block.
- To control the volume by using the `DKAUDIO` (play audio) operation, the device must support the 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 mail summary file (.msf)
  - play audio track index
  - read table of contents (TOC)

Note: Only the International Organization for Standardization (ISO) file system (read-only ISO 9660), Universal Disk Format (UDF) file system Version 2.01, or earlier, are supported on USB devices for the AIX operating system. However, you can create a system backup or data archival on the drives by using the **mksysb**, **tar**, **cpio**, **backup**, or **restore** commands. You can also use the **dd** command to add the ISO images to the drives.

To use the USB flash drive, RDX, CD-ROM, DVD-RAM, and Blu-ray read-only devices, install the following device package:

```
devices.usbif.08025002
```

The AIX operating system does not support plug-and-play feature for USB devices. To make a flash drive, RDX, CD-ROM, Blu-ray, or DVD-RAM drive available to the AIX users, a root user must connect the drive to a system USB port and run the following command:

```
# cfgmgr -l usb0
```

Note: Use caution when you remove the flash drives from ports. If the drives are not properly closed or unmounted before you remove the drives, the data on the drives can be corrupted.

After you remove the drives, the drives remain in the available state in the Object Data Manager (ODM) until the root user runs the following command:

```
# rmdev -l usbmsn
```

or

```
#rmdev –l cdn
```

When a drive is in the available state, you can reconnect the drive to the system, and the drive can be remounted or reopened. If a drive is disconnected from a system USB port while it is still open for a user, that drive is not reusable until you close and reopen it.

AIX Version 6.1 with the 6100-06 Technology Level recognizes and configures USB attached Blu-ray drives as read-only. The AIX operating system does not support the write operation to CD, DVD, or Blu-ray media that are present in the USB Blu-ray drive. Although the write operation is not prevented (if the drive is write-capable), no support is provided for any issues that are encountered during the write operation.

The capability of the AIX operating system to operate on USB original equipment manufacturer (OEM) flash drive, Blu-ray, and optical devices is validated against a sample of industry standard OEM USB devices that are compliant with the USB standards. You might encounter issues with certain USB devices that are not compliant and the AIX operating system does not provide any support for those issues.

Related information:

Options to the openx subroutine

usbms special file

USB subsystem overview

ioctl subroutine

Error Conditions for USB Mass Storage Client Device Driver

Possible *errno* values for `ioctl`, `open`, `read`, and `write` subroutines when you use the `scsidisk` device driver include the following values:

Value	Description
EACCES	Indicates one of the following conditions: • An attempt was made to open a device that is currently open in the `Diagnostic` or `Exclusive Access` mode. • An attempt was made to open a `Diagnostic` mode session on a device that is already open. • You attempted to run a subroutine other than an `ioctl` or `close` subroutine while in `Diagnostic` mode. • A `DKIOLCMD` operation was attempted on a device that is not in the `Diagnostic` mode. • A `DK_CD_MODE` `ioctl` subroutine operation was attempted on a device that is not in the `Exclusive Access` mode.
EBUSY	Indicates one of the following conditions: • An attempt was made to open a session in the `Exclusive Access` mode on a device that is already opened. • The target device is reserved by another initiator.
EFAULT	Indicates an invalid user address.
EFORMAT	Indicates that the target device has unformatted media or the media is in an incompatible format.
EINPROGRESS	Indicates that 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 *n*byte 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 `DKIORDSE` `ioctl` subroutine operation. • The data buffer length exceeded the maximum value that is defined in the `devinfo` structure for a `DKIORDSE` 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 invalid configuration command is provided. • 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 value that is 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 indicates an unrecoverable hardware error.

Value	Description
EMEDIA	Indicates one of the following circumstances: • The target device indicates an unrecoverable media error. • The media was changed.
EMFILE	Indicates that 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 that there is no media 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 hard disk.
EPERM	Indicates that the attempted subroutine requires appropriate authority.
ESTALE	Indicates that 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 that an I/O operation exceeded the specified timer value.
EWRPROTECT	Indicates one of the following circumstances: • An open operation that requires read/write mode was attempted on a read-only media. • A write operation was attempted to a read-only media.

Reliability and serviceability information

USB hard disk, flash drive, RDX devices, CD-ROM drives, and read/write optical drives return the following errors:

Error	Description
ABORTED COMMAND	Indicates that the device ended the command.
ADAPTER ERRORS	Indicates that the adapter returned an error.
GOOD COMPLETION	Indicates that the command completed successfully.
HARDWARE ERROR	Indicates that an unrecoverable hardware failure occurred when the command was run or during a self-test.
ILLEGAL REQUEST	Indicates an invalid command or command parameter.
MEDIUM ERROR	Indicates that the command ended with an unrecoverable media error condition.
NOT READY	Indicates that the logical unit is offline or the media is missing.
RECOVERED ERROR	Indicates that the command was successful after some recovery was applied.
UNIT ATTENTION	Indicates that the device is reset or the power is turned on.

The fields that are defined in the error record template for hard disk, flash drive, RDX, CD-ROM, and read/write optical media errors are logged as per the following structure:

```
/* Bulk transfer cmd and status blocks */
typedef struct mstor_cbw {
    uint32_t cbw_signature;     /* Always "USBC" little endian */
    uint32_t cbw_tag;              /* Command identification  */
    fld32_t cbw_dlen;              /* Data length  */
    uchar cbw_flags;               /* Indicates data in or data out */
    uchar cbw_lun;                 /* Logical unit number, 0-15  */
    uchar cbw_cblen;               /* Significant bytes of the cmd blk */
    uchar cbw_cb[16];              /* Command block itself  */
    uchar cbw_rsvd;
} mstor_cbw_t;
```

```
/* For error logging */
struct mstor_err_rec {
    struct err_rec0 log;
    uint cmd_error;
    mstor_cbw_t cbw;
    char sense_data[128];
};

LABEL:          DISK_ERRx
IDENTIFIER:     xxxxxxxx

Date/Time:       Wed Aug  4 11:40:43 CDT 2010
Sequence Number: 80
Machine Id:      00000A2AD400
Node Id:         node10
Class:           H
Type:            PERM
Resource Name:   usbms0
Resource Class:  usbms
Resource Type:   0806500b
Location:        U78A5.001.WIH00AD-P1-T1-L1-L2-L3

Description
Probable Causes
User Causes
Failure Causes

SENSE DATA
1111 2222 2222 3333 3333 4444 4444 5566 LLCC CCCC CCCC CCCC CCCC CCCC CCCC CCCC
CCRR SSSS KKSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS
SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS
SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS
SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSSS SSOO SSNN

Data Representation Legend
--------------------------
   cmd_error              1    Command Error Value
  (cmd_error values can be negative which are logged as 2's complement.
   For these USB specific error values refer below or /usr/include/sys/usbdi.h.
  For error values which are positive Please refer to /usr/include/sys/errno.h file for error description)

Bulk transfer Command and Status Blocks
   cbw_signature         2    Always .USBC. in ASCII – "5553 4243"
   cbw_tag                  3    Command Identification
   cbw_dlen                4    Data Length
   cbw_flags               5    Indicates Data IN or OUT
   cbw_lun                  6    LUN Id
   cbw_cblen               L    CDB (Command Descriptor Bytes) length
   cbw_cb                  C    CDB – SCSI/ATAPI Command Set
   cbw_rsvd                R    Reserved

Sense data
   Sense data             S
   Sense key              K
   ASC                    c
   ASCQ                   q
   Driver Open Count      O
   Location               N   Device Driver log location
```

Error record values for media errors

Value	Description
Comment	Indicates hard disk, flash drive, RDX, CD-ROM, or read/write optical media error.
Class	Equals a value of `H` that indicates a hardware error.
Report	Equals a value of True that indicates this error must be included when an error report is generated.
Log	Equals a value of True that indicates an error log entry must be created when this error occurs.
Alert	Equals a value of False that indicates this error cannot have an alert.
Err_Type	Equals a value of Perm that indicates a permanent failure.
Err_Desc	Equals a value of 1312 that indicates a disk operation failure.
Prob_Causes	Equals a value of 5000 that indicates media.
User_Causes	Equals a value of 5100 that indicates the media is defective.
User_Actions	Equals the following values: • 1601, which indicates the removable media must be replaced and tried again. • 00E1, which instructs to 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 that the removable media must be replaced and tried again. • 00E1, which instructs to 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 `mstor_err_rec` structure. The `err_rec` field is defined in the `/usr/include/sys/errids.h` file. The `Detail_Data` field follows the same legend as mentioned in the preceding structure example.

Refer to the Small Computer System Interface (SCSI) specifications for the format of the request-sense data for a particular device.

Error record values for hardware errors

The fields that are defined in the error record template for hard disk, CD-ROM, and read/write optical hardware errors and for hard-aborted command errors are listed in the following table:

Value	Description
Comment	Indicates hard disk, flash drive, RDX, CD-ROM, or read/write optical hardware error.
Class	Equals a value of H that indicates a hardware error.
Report	Equals a value of True that indicates this error must be included when an error report is generated.
Log	Equals a value of True that indicates an error log entry must be created when this error occurs.
Alert	Equals a value of False that indicates this error cannot be alerted.
Err_Type	Equals a value of Perm that indicates a permanent failure.
Err_Desc	Equals a value of 1312 that indicates a disk operation failure.
Prob_Causes	Equals a value of 6310 that 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.

Value	Description
Fail_Actions	Equals a value of 00E1 that indicates problem-determination procedures must 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 `mstor_err_rec` structure. The `err_rec` field is defined in the `/usr/include/sys/errids.h` file. It follows the same legend as mentioned in the preceding structure example.

Error record values for adapter-detected hardware failures

The following fields are defined in the error record template for hard disk, CD-ROM, and read/write optical media errors and for adapter-detected hardware errors:

Value	Description
Comment	Indicates adapter-detected hard disk, flash drive, RDX, CD-ROM, or read/write optical hardware failure.
Class	Equals a value of H that indicates a hardware error.
Report	Equals a value of True that indicates that this error must be included when an error report is generated.
Log	Equals a value of True that indicates that an error-log entry must be created when this error occurs.
Alert	Equals a value of False that indicates this error cannot be alerted.
Err_Type	Equals a value of Perm that indicates a permanent failure.
Err_Desc	Equals a value of 1312 that 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 that indicates that the problem-determination procedures must 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 `mstor_err_rec` structure. The `err_rec` field is defined in the `/usr/include/sys/errids.h` file. It follows the same legend as mentioned in the preceding structure example.

Error record values for recovered errors

The following fields are defined in the error record template for hard disk, CD-ROM, and read/write optical media recovered errors:

Item	Description
Comment	Indicates hard disk, CD-ROM, or read/write optical recovered error.
Class	Equals a value of H that indicates a hardware error.
Report	Equals a value of True that indicates this error must be included when an error report is generated.
Log	Equals a value of True that indicates an error log entry must be created when this error occurs.
Alert	Equal to a value of False that indicates this error cannot be alerted.
Err_Type	Equals a value of Temp that indicates a temporary failure.
Err_Desc	Equals a value of 1312 that 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 that indicates that the media is defective.
User_Actions	Equals the following values: • 0000, which indicates that the problem-determination procedures must be performed • 1601, which indicates that the removable media must be replaced and tried again
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 that the removable media must be replaced and tried again • 00E1, which performs 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 `mstor_err_rec` structure. The `err_rec` field is defined in the `/usr/include/sys/errids.h` file. It follows the same legend as other errors.

Error record values for unknown errors

The following fields are defined in the error record template for hard disk, CD-ROM, and read/write optical media unknown errors:

Value	Description
Comment	Indicates hard disk, CD-ROM, or read/write optical unknown failure.
Class	Equals a value of H that indicates a hardware error.
Report	Equals a value of True that indicates this error must be included when an error report is generated.
Log	Equals a value of True that indicates an error log entry must be created when this error occurs.
Alert	Equal to a value of False that indicates this error cannot be alerted.
Err_Type	Equals a value of Unkn that indicates the type of error is unknown.
Err_Desc	Equals a value of FE00 that 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.

Value	Description
Inst_Actions	None.
Fail_Causes	Equals a value of FFFF that indicates the failure causes are unknown.
Fail_Actions	Equals the following values: • 00E1, which performs problem determination procedures • 1601, which indicates the removable media must be replaced and tried again
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 `mstor_err_rec` structure. The `err_rec` field is defined in the `/usr/include/sys/errids.h` file. It follows the same legend as other errors.

Special Files

The `usbcd` USB client device driver uses raw and block special files for its functions. The special files that are used by the `usbcd` device driver are listed by the type of device in the following table:

Table 1. Special files for the usbcd device driver

Device	Special file	Description
Hard disk, flash drive, RDX devices	`/dev/rusbms0`, `/dev/rusbms1`, ..., `/dev/rusbms`*n*	Provides an interface for USB client device drivers to access character (raw I/O access and control functions).
	`/dev/usbms0`, `/dev/usbms1`, ..., `/dev/usbms`*n*	Provides an interface for USB client device drivers to access block I/O.
CD-ROM, DVD-RAM, Blu-ray read-only devices:	`/dev/rcd0`, `/dev/rcd1`, ..., `/dev/rcd`*n*	Provides an interface for USB client device drivers to access character (raw I/O access and control functions).
	`/dev/cd0`, `/dev/cd1`, ..., `/dev/cd`*n*	Provides an interface for USB client device drivers to access block I/O.

Note: The prefix `r` on a special file name indicates that the drive is accessed as a raw device rather than a block device. Performing raw I/O with a hard disk, flash drive, RDX, CD-ROM, or read/write optical drive requires all data transfers to be in multiples of the device block size. Also, all the **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:

open and openx subroutines

cd subroutine

Required USB Adapter Driver ioctl Commands

RDX USB External Dock (1104 and EU04) and RDX Removable Disk Drives (1106, 1107, EU01, EU08, and EU15)

USB Mouse Client Device Driver

Purpose

Supports the Universal Serial Bus (USB) mouse device.

Syntax

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

Description

The USB mouse client device driver consists of a back end that interfaces with the USB system driver (USBD) and a front end that interfaces with an AIX application such as the X server application. The client driver has no knowledge of the underlying USB adapter hardware. Instead, the client driver sends control requests to the USB mouse through the USBD and receives input events through the USBD. The USB mouse client driver supports the attachment of multiple USB mouse devices. Each device is enumerated in the Object Data Manager (ODM) and marked available. The client driver treats all the mouse devices as a single logical device. Input events from all the devices are sent to a single input ring.

A device special file is created for each USB mouse device. Until there is at least one USB mouse device that is marked available, an application (typically the X server application) can open any one of the USB mouse special files because the client driver ignores the minor number specification. A USB mouse device that is added and configured after the open operation is automatically added to the open set. The device special files (for example, /dev/mouse0, /dev/mouse1, and so on) are created for each USB mouse device.

Special treatment for the mouse is provided by the USBD configuration method because of the strict configuration and ordering rules of the graphics subsystem. When a USB host controller and a graphics adapter are marked available, and no existing mouse is present, the USBD ensures that at least one USB mouse instance is defined. The USB mouse client driver uses the USBD_OPEN_DEVICE_EXT ioctl operation to open the device that generates a valid handle even when there is no USB mouse that is attached to the system. The EAGAIN error code is returned by the USBD_OPEN_DEVICE_EXT operation if there is no mouse device and the client driver treats the device as disconnected. When you plug in a USB mouse, a reconnect call back operation is made to the mouse client by the USBD and the device is initialized for the input events to flow to the X server application.

The following input device driver ioctl operations are used for the USB mouse operations:

Operation	**Description**
IOCINFO	Returns a devinfo structure, which is defined in the sys/devinfo.h header file, that describes the device. The first field of the structure (devtype) is set to DD_INPUT; the rest of the structure is set to zero.
MQUERYID	Queries mouse device identifier.
MREGRING	Registers input ring.
MREGRINGEXT	Registers extended 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.

Related information:

mouse special file

USB Tape Client Device Driver

Purpose

Supports the Universal Serial Bus (USB) protocol for sequential access tape device driver.

Syntax

```
#include <sys/devinfo.h>
#include <sys/usb.h>
#include <sys/tape.h>
#include <sys/usbdi.h>
```

Device-dependent subroutines

Most of the tape operations are implemented by using the open, close, read, and write subroutines. However, the openx subroutine must be used if the device must be opened in the Diagnostic mode.

open and close subroutines

The openx subroutine is primarily used for the diagnostic commands and utilities. Appropriate authority is required for to run the subroutine. If you run the openx subroutine without the required authority, the subroutine returns a value of -1 and sets the *errno* global variable to a value of EPERM.

The openx subroutine enables the Diagnostic mode for the device driver and disables command-retry logic. This action allows the ioctl operations that perform special functions that are associated with diagnostic processing. The openx subroutine can also force-open and retain reservations.

The open subroutine applies a reservation policy that is based on the Object Data Manager (ODM) reserve_policy attribute. The USB tape devices might not support Small Computer System Interface (SCSI) reservation command and therefore, these commands might be ignored.

The *ext* parameter that is passed to the openx subroutine selects the operation to be used for the target device. The /usr/include/sys/scsi.h file defines the possible values for the *ext* parameter.

The *ext* parameter can contain any logical combination of the following flag values:

Item	Description
SC_FORCED_OPEN	Forces access to a device by removing any type of reservation on the device that can inhibit access. The type of action to remove the reservation depends upon the specific type of the established reservation. If this flag is specified, a mass storage reset command is issued for a USB tape, which is a mass storage bulk device.
SC_DIAGNOSTIC	Places the selected device in the Diagnostic mode. This mode is singularly entrant. It means when a device is in the Diagnostic mode, SCSI operations are performed during the open or close operations, and error logging is disabled. In the Diagnostic mode, only the close and ioctl 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 the Diagnostic mode only if the target device is not currently opened. If you open a device in the 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.

ioctl subroutine

The following ioctl operations are supported on USB tape devices:

Operation	Description

IOCINFO	Populates the devinfo argument that is passed by the caller with the following values:
	`devinfo.devtype = DD_SCTAPE;`
	`devinfo.flags = 0;`
	`devinfo.devsubtype = 0x00;`
	`devinfo.un.scmt.type = DT_STREAM;`
	`devinfo.un.scmt.blksize = *Block Size Set for the Tape Device*;`

Operation	Description
STIOCTOP	Specifies the address of a stop structure that is defined in the `src/bos/usr/include/sys/tape.h` file. The operation that is found in the `st_op` field in the stop structure is run *st_count* times, except for rewind, erase, and retention operations. This `ioctl` command supports the following operations with the respective implementation details: **STREW** Issues the `REWIND` command to the tape device to rewind the tape. **STERASE** Issues the `SCSI ERASE` command to erase the contents of the tape media. Erase is not allowed with a read-only tape. **STRETEN or STINSRT** Issues the `SCSI LOAD` command with `Load` and `Reten` bits that are set in byte 4 of the command. **STWEOF** Writes the end-of-file mark to the tape. The `Write End-of-Filemark` operation is not allowed with a read-only tape. **STDEOF** Disables the end-of-tape checking command. **STFSF** Issues the `Forward Space File` command. The `st_count` field specifies the number of file marks that the tape advances. **STFSR** Issues the `Forward Space Record` command. The `st_count` field is the number of records that the tape advances. **STRSF** Issues the `Reverse Space File` command. The `st_count` field is the number of file marks that the tape reverses. **STRSR** Issues the `Reverse Space Record` command. The `st_count` field is the number of records that the tape reverses. **STOFFL or STEJECT** Ejects the tape from the tape drive. This operation issues the `SCSI LOAD` command with `Load` bit in byte 4 of Command Descriptor Block (CDB) that is set to zero. **STIOCHGP** Defines the `ioctl` command to dynamically change the block size for this tape device. The block size is changed for the length of the open operation and is returned to the original values on the next open operation. The tape is forced to BOT (beginning-of-tape) when this operation is performed. The parameter to this `ioctl` command specifies the address of a `stchgp` structure that is defined in the `src/bos/usr/include/sys/tape.h` file. The `st_blksize` field in the structure specifies the block size value to be set.

Operation	Description
STIOCTOP (continued)	**STIOCMD**

When the device is successfully opened, the STIOCMD operation issues an SCSI command to the specified tape device.

The SCSI status byte and the adapter status bytes are returned through the **arg** parameter that contains the address of a scsi_iocmd structure. This structure is defined in the /usr/include/sys/scsi_buf.h file. The STIOCMD operation receives the SCSI command in the scsi_cdb section of the scsi_iocmd structure and issues it to the USB tape device. If the STIOCMD operation fails, the subroutine returns a value of -1 and sets the *errno* global variable to a nonzero value. In this case, the caller must evaluate the returned status bytes to determine the cause of operation failure and the recovery actions.

The version, command_length, and timeout_value values that are passed by the user is validated and error value EINVAL is returned if they are not valid.

If you transfer more than 1 MB of the maximum I/O transfer size, the subroutine returns a value of -1 and sets the *errno* global variable to a value of EINVAL.

On a check condition, the following error status values are set in the sc_passthru structure:

```
status_validity  = SC_SCSI_ERROR
scsi_bus_status = SC_CHECK_CONDITION
adap_set_flags will have SC_AUTOSENSE_DATA_VALID flag set.
```

The following example is a pseudo code to issue the STIOCMD operation to the USB tape to issue an INQUIRY SCSI command:

```
    struct scsi_iocmd cmd;
    char inq_data[255];
    char sense_data[255];
..
    fd = open("/dev/rmt0", O_RDWR);
..
    memset(&cmd, '\0', sizeof(struct sc_passthru));
    cmd.version = SCSI_VERSION_1;
    cmd.timeout_value = 30;
    cmd.command_length = 6;
    cmd.autosense_length = 255;
    cmd.autosense_buffer_ptr = &sense_data[0];
    cmd.data_length = 0xFF;
    cmd.buffer = inq_data;

   cmd.flags = B_READ;

   cmd.scsi_cdb[0] = SCSI_INQUIRY;
    cmd.scsi_cdb[1] = (0x00 | vpd);  /* Standard Inquiry – vpd=1
                 for Extended Inquiry */
    cmd.scsi_cdb[2] = page_code;    /* Page Code – valid if vpd=1 */
    cmd.scsi_cdb[3] = 0x00;
    cmd.scsi_cdb[4] = 0xFF;
    cmd.scsi_cdb[5] = 0x00;

   if ((rc=ioctl(fd, STIOCMD, &cmd)) != 0){
         if (cmd.adap_set_flags & SC_AUTOSENSE_DATA_VALID) {
        /* look at sense data */
         } /* end SC_AUTOSENSE_DATA_VALID */

         printf("STPASSTHRU: Ioctl FAIL errno %d\n",errno);
         printf("status_validity: %x, scsi_status: %x, adapter_status:%x\n",
      cmd.status_validity, cmd.scsi_bus_status, cmd.adapter_status);
         printf("Residual: %x\n", cmd.residual);
         exit(-1);
     } else {
         printf("STPASSTHRU : Ioctl PASS\n");
         printf("status_validity: %x, scsi_status: %x, adapter_status:%x\n",
      cmd.status_validity, cmd.scsi_bus_status, cmd.adapter_status);     }
```

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>STPASSTHRU</td><td>

Takes the SCSI command in the scsi_cdb section of the sc_passthru structure and issues it to the USB tape driver. This operation is similar to the STIOCMD ioctl operation with the only exception of additional informative fields in the sc_passthru structure that provides more information on the error.

The following example is a pseudo code to issue the STPASSTHRU operation to the USB tape to issue an INQUIRY SCSI command:

```
    struct sc_passthru cmd;
    char inq_data[255];
    char sense_data[255];
..
    fd = open("/dev/rmt0", O_RDWR);
..
    memset(&cmd, '\0', sizeof(struct sc_passthru));
    cmd.version = SCSI_VERSION_1;
    cmd.timeout_value = 30;
    cmd.command_length = 6;
    cmd.autosense_length = 255;
    cmd.autosense_buffer_ptr = &sense_data[0];
    cmd.data_length = 0xFF;
    cmd.buffer = inq_data;

   cmd.flags = B_READ;

   cmd.scsi_cdb[0] = SCSI_INQUIRY;
    cmd.scsi_cdb[1] = (0x00 | vpd);  /* Standard Inquiry - vpd=1
                                 for Extended Inquiry */
    cmd.scsi_cdb[2] = page_code;      /* Page Code - valid if vpd=1 */
    cmd.scsi_cdb[3] = 0x00;
    cmd.scsi_cdb[4] = 0xFF;
    cmd.scsi_cdb[5] = 0x00;

   if ((rc=ioctl(fd, STPASSTHRU, &cmd)) != 0){
       if (cmd.adap_set_flags & SC_AUTOSENSE_DATA_VALID) {
             /* look at sense data */
         } /* end SC_AUTOSENSE_DATA_VALID */

         printf("STPASSTHRU: Ioctl FAIL errno %d\n",errno);
         printf("status_validity: %x, scsi_status: %x, adapter_status:%x\n",
       cmd.status_validity, cmd.scsi_bus_status, cmd.adapter_status);
         printf("Residual: %x\n", cmd.residual);
         exit(-1);
     } else {
         printf("STPASSTHRU : Ioctl PASS\n");
         printf("status_validity: %x, scsi_status: %x, adapter_status:%x\n",
       cmd.status_validity, cmd.scsi_bus_status, cmd.adapter_status);
     }
```

</td></tr>
</table>

Related information:

Options to the openx subroutine

rmt special file

open subroutine

write subroutine

Error Conditions for USB Tape Client Device Driver

In addition to the listed errors, the ioctl, open, read, and write subroutines for USB tape device are unsuccessful in the following circumstances:

Value	Description
EAGAIN	Indicates that an attempt is made to open a device, which is already open.
EBUSY	Indicates that the target device is reserved by another initiator.
EINVAL	• Indicates that the `O_APPEND` value is supplied as the mode in which the device is to be opened. • Indicates that the *n*`byte` parameter that is supplied by a `read` or `write` operation is not a multiple of the block size. • Indicates that a parameter to an `ioctl` operation is not valid. • Indicates that the requested `ioctl` operation is not supported on the current device.
EIO	• Indicates that the tape drive is reset or the tape is changed. This error is returned during the open operation if the tape is positioned beyond the beginning of the tape upon closing as a result of the previous operation to the tape. • Indicates that the device cannot space forward or reverse the number of records that is specified by the `st_count` field before it encounters an end-of-media (EOM) or a file mark.
EMEDIA	Indicates an open operation is attempted for an adapter that already has the maximum permissible number of opened 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, which has already reached EOM.
EPERM	Indicates that the subroutine requires appropriate authority.
ETIMEDOUT	Indicates that a command has timed out.
EWRPROTECT	• Indicates that an open operation is attempted for the read/write mode on a read-only tape. • Indicates that an ioctl operation, which affects the media, was attempted on a read-only tape.

Reliability and serviceability information

The following errors are returned from the tape devices:

Error	Description
ABORTED COMMAND	Indicates that the device ended the command.
BLANK CHECK	Indicates that a read command encountered a blank tape.
DATA PROTECT	Indicates that a write operation was attempted on a write-protected tape.
GOOD COMPLETION	Indicates the command completed successfully.
HARDWARE ERROR	Indicates that an unrecoverable hardware failure occurred during the command execution or during a self-test.
ILLEGAL REQUEST	Indicates an invalid command or an invalid command parameter.
MEDIUM ERROR	Indicates that the command ended with an unrecoverable media error condition. This condition can be caused by a tape flaw or a dirty head.
NOT READY	Indicates that the logical unit is offline.
RECOVERED ERROR	Indicates the command is successful after some recovery operations were applied.
UNIT ATTENTION	Indicates that the device is reset or the power is turned on.

Medium, hardware, and unsuccessful command errors from the preceding list must be logged every time they occur. The `ABORTED COMMAND` error might 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.

```
/* Bulk transfer cmd and status blocks */
typedef struct mstor_cbw {
    uint32_t cbw_signature;     /* Always "USBC" little endian */
    uint32_t cbw_tag;           /* Command identification  */
    fld32_t cbw_dlen;           /* Data length  */
    uchar cbw_flags;            /* Indicates data in or data out */
    uchar cbw_lun;              /* Logical unit number, 0-15  */
    uchar cbw_cblen;            /* Significant bytes of the cmd blk */
    uchar cbw_cb[16];           /* Command block itself  */
    uchar cbw_rsvd;
} mstor_cbw_t;

/* For error logging */
struct usbtape_err_rec {
    struct err_rec0 log;
    uint cmd_error;
    mstor_cbw_t cbw;
    char sense_data[168];
    uint dd1;                    /* reserved for dd use */
    uint dd2;                    /* reserved for dd use */
    uint dd3;                    /* reserved for dd use */
    uint dd4;                    /* reserved for dd use */
    uint dd5;                    /* reserved for dd use */
    uint dd6;                    /* reserved for dd use */
    uint dd7;                    /* reserved for dd use */
    uint dd8;                    /* reserved for dd use */
};

LABEL:          SC_TAPE_ERRx
IDENTIFIER:     xxxxxxxx
Date/Time:       Thu Mar 12 05:20:27 CDT 2009
Sequence Number: 3829
Machine Id:      0000097AD400
Node Id:        sitar04
Class:           H
Type:            PERM
Resource Name:   rmt0
Resource Class:  tape
Resource Type:   0806500c
Location:

Description
Probable Causes
Failure Causes

       Recommended Actions

Detail Data

SENSE DATA
1111 1111 2222 2222 3333 3333 4444 4444 5566 7788 8888 8888 8888 8888 8888 8888
8888 8899 aaaa kkaa aaaa aaaa aaaa aaaa ccqq aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa bbbb bbbb cccc cccc dddd dddd


Data Representation Legend
--------------------------
   cmd_error             1    Command Error Value
(cmd_error values can be negative which are logged as 2's complement.
  For these USB specific error values refer below or /usr/include/sys/usbdi.h.
  For error values which are positive Please refer to /usr/include/sys/errno.h file for error description)

Bulk transfer Command and Status Blocks
   cbw_signature         2    Always .USBC. in ASCII – "5553 4243"
   cbw_tag               3    Command Identification
   cbw_dlen             4    Data Length
   cbw_flags            5    Indicates Data IN or OUT
   cbw_lun              6    LUN Id
   cbw_cblen            7    CDB (Command Descriptor Bytes) length
   cbw_cb               8    CDB
   cbw_rsvd             9    Reserved

Sense data
   Sense data           a
   Sense key            k
   ASC                  c
   ASCQ                 q
   Read Transfer Count  b   In bytes
   Write Transfer Count c   In bytes
   Location             d   Device Driver log location
```

Note: Device-related adapter errors are logged every time the errors occur.

Error record values for tape device errors

The following table lists the fields that are defined in the error record template for tape device errors:

Error ID	Description
SC_TAPE_ERR1	Permanent tape error. This error is logged when tape medium error is encountered.
SC_TAPE_ERR2	Permanent tape hardware error. This error is logged when tape hardware error is encountered or command is aborted by the drive and all attempts to resolve the error have failed.
SC_TAPE_ERR3	Temporary tape drive failure. This error is not logged in Universal Serial Bus (USB) tape driver.
SC_TAPE_ERR4	Permanent tape drive failure. This error is logged when adapter failure is detected and all attempts have failed.
SC_TAPE_ERR5	Unknown tape error. This error is logged when tape returns a check condition but the sense data does not contain valid information.
SC_TAPE_ERR6	Temporary tape operation error. Tape drive needs to be cleaned.
SC_TAPE_ERR7	Informational error. Remote Access Service (RAS) related error logs due to internal driver sanity check failures.
SC_TAPE_ERR8	Temporary tape drive failure. This error is not logged in USB tape driver.

Error record values for tape device media errors

The following table lists the fields that are defined in the error record template for tape device media errors:

Item	Description
Comment	The tape media error.
Class	A value of H that indicates a hardware error.
Report	A value of True that indicates this error must be included when an error report is generated.
Log	A value of True that indicates an error log entry must be created when this error occurs.
Alert	A value of False that indicates this error cannot be alerted.
Err_Type	A value of Perm that indicates a permanent failure.
Err_Desc	A value of 1332 that indicates a tape operation failure.
Prob_Causes	A value of 5003 that indicates tape media.
User_Causes	A value of 5100 that indicates an error with the tape device and a value of 7401 that indicates an error with the defective media.

Item	Description
User_Actions	A value of 1601 that indicates that the removable media must be replaced and the operation must be tried again. Or, it equals a value of 0000 that indicates that problem determination procedures must be performed.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	A value of 5003 that indicates tape media.
Fail_Actions	A value of 1601 that indicates that the removable media must be replaced and the operation must be tried again. Or, it equals a value of 0000 that indicates that problem determination procedures must 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.

Error record values for tape or hardware aborted command errors

The following fields in the err_hdr structure are defined in the /usr/include/sys/erec.h file for hardware errors and aborted command errors:

Item	Description
Comment	A value of tape hardware or aborted command error.
Class	A value of H that indicates a hardware error.
Report	A value of True that indicates this error must be included when an error report is generated.
Log	A value of True that indicates an error log entry must be created when this error occurs.
Alert	A value of FALSE that indicates this error cannot be alerted.
Err_Type	A value of Perm that indicates a permanent failure.
Err_Desc	A value of 1331 that indicates a tape drive failure.
Prob_Causes	A value of 6314 that indicates a tape drive error.
User_Causes	None.
User_Actions	A value of 0000 that indicates that problem determination procedures must be performed.
Inst_Actions	None.
Fail_Causes	A value of 5003 that indicates the failure case is the tape and a value of 6314 that indicates the failure case is the tape drive.
Fail_Actions	A value of 0000 that indicates that problem determination procedures must 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 usbtape_err_rec structure describes information that is contained in the Detail_Data field.

Error record values for tape-recovered error threshold exceeded

The following fields are defined in the err_hdr structure that are defined in the /usr/include/sys/erec.h file for recovered errors that have exceeded the threshold counter:

Item	Description
Comment	Indicates that the threshold for the tape-recovered errors is exceeded.
Class	A value of H that indicates a hardware error.
Report	A value of True that indicates this error must be included when an error report is generated.
Log	A value of True that indicates an error-log entry must be created when this error occurs.
Alert	A value of False that indicates this error cannot be alerted.
Err_Type	A value of TEMP that indicates a temporary failure.
Err_Desc	A value of 1331 that indicates a tape drive failure.
Prob_Causes	A value of 6314 that indicates the probable cause is the tape drive.
User_Causes	A value of 5100 that indicates the media is defective and a value of 7401 that indicates the read/write head is dirty.
User_Actions	A value of 1601 that indicates that the removable media must be replaced and the operation must be tried again. Or, it equals a value of 0000 that indicates that problem determination procedures must be performed.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	A value of 5003 that indicates the failure cause is the tape and a value of 6314 that indicates the failure cause is tape drive.
Fail_Actions	A value of 0000 that indicates problem-determination procedures must 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. 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 `usbtape_err_rec` structure describes information contained in the `Detail_Data` field.

Error record values for tape USB adapter-detected errors

The following fields in the `err_hdr` structure are defined in the `/usr/include/sys/erec.h` file for adapter-detected errors:

Item	Description
Comment	A tape Fibre Channel adapter-detected error.
Class	A value of H that indicates a hardware error.
Report	A value of True that indicates this error must be included when an error report is generated.
Log	A value of True that indicates an error log entry must be created when this error occurs.
Alert	A value of FALSE that indicates this error cannot be alerted.
Err_Type	A value of PERM that indicates a permanent failure.
Err_Desc	A value of 1331 that indicates a tape drive failure.
Prob_Causes	The values of 3300 that indicates adapter failure and a value of 6314 that indicates tape drive failure.
User_Causes	None.
User_Actions	A value of 0000 that indicates that problem determination procedures must be performed.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	A value of 3300 that indicates adapter failure and a value of 6314 that indicates tape drive failure.

Item	Description
Fail_Actions	A value of 0000 that indicates problem-determination procedures must be performed.

The Detail_Data field contains the command type and adapter status. This field is contained in the err_rec structure that is defined by the /usr/include/sys/err_rec.h file. Request-sense information is not available with this type of error. The usbtape_err_rec structure describes information contained in the Detail_Data field.

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.

Item	Description
Comment	Indicates that the tape drive needs cleaning.
Class	A value of H that indicates a hardware error.
Report	A value of True that indicates this error must be included when an error report is generated.
Log	A value of True that indicates an error log entry must be created when this error occurs.
Alert	A value of FALSE that indicates this error cannot be alerted.
Err_Type	A value of TEMP that indicates a temporary failure.
Err_Desc	A value of 1332 that indicates a tape operation error.
Prob_Causes	A value of 6314 that indicates that the probable cause is the tape drive.
User_Causes	A value of 7401 that indicates a dirty read/write head.
User_Actions	A value of 0000 that indicates that problem determination procedures must be performed.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	A value of 6314 that indicates that the cause is the tape drive.
Fail_Actions	A value of 0000 that indicates problem-determination procedures must be performed.

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 that is defined by the /usr/include/sys/errids.h file. The usbtape_err_rec structure describes information contained in the Detail_Data field.

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, which are detected by the tape device driver, are never seen at the tape drive.

The err_hdr structure for unknown errors describes the following fields:

Item	Description
Comment	A tape unknown error.
Class	All error classes.
Report	A value of True that indicates this error must be included when an error report is generated.
Log	A value of True that indicates an error log entry must be created when this error occurs.
Alert	A value of FALSE that indicates this error cannot be alerted.
Err_Type	A value of UNKN that indicates the type of error is unknown.
Err_Desc	A value of 0xFE00 that indicates the error description is unknown.

Item	Description
Prob_Causes	Specifies the following values: • 3300, which indicates a tape drive failure • 5003, which indicates a tape failure • 6314, which indicates an adapter failure
User_Causes	None.
User_Actions	None.
Inst_Causes	None.
Inst_Actions	None.
Fail_Causes	A value of 0xFFFF that indicates the causes for failure are unknown.
Fail_Actions	A value of 0000 that indicates that problem-determination procedures must 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 usbtape_err_rec structure describes information that is contained in the Detail_Data field.

Refer to the SCSI specification for the applicable device for the format of the particular request-sense information.

Related information:

USB subsystem overview

Understanding the Execution of Initiator I/O Requests

USB Error Recovery

Managing tape drives

USBD Protocol Driver

Purpose

Supports the USB system driver (USBD) protocol.

Syntax

```
#include <sys/usb.h>
#include <sys/usbdi.h>
#include <sys/hubClass.h>
#include <sys/hidClass.h>
```

Description

The USBD protocol driver is the layer between the host controller the and client drivers. The /dev/usb0 special file provides interface to allow communication between the host controller and the client drivers. This driver is responsible for the device communication to appropriate host controller. The device connection, disconnection, and re-connection are performed at this level. There is no parent for this device and the device's CuDv entry is created by the /usr/lib/methods/startusb script that is invoked from the ConfigRules field.

The /usr/lib/drivers/usb/usbd driver implements the USB protocol and the /usr/lib/methods/cfgusb file is the usbd file's configuration method. The USB protocol driver updates the speed ODM attribute that is specific to each individual USB devices. The speed is updated when the USB devices are enumerated during the AIX configuration.

Device-dependant subroutines

The USBD protocol driver supports only the open, close, and ioctl subroutines. The read and write subroutines are not supported.

open and close subroutines

The open subroutine associates a specific device number that is passed in as a parameter to the open system call with the internal adapter device structure. If it finds an adapter structure, it verifies that the corresponding adapter device is configured and sets the state as open. Otherwise, the subroutine returns an error.

ioctl subroutine

The ioctl operations for USBD protocol drivers are exposed to kernel and user environments.

USBD ioctl operations

The following USBD ioctl operations are exposed to kernel threads that are used to open a specific USB logical device:

- USBD_OPEN_DEVICE
- USBD_OPEN_DEVICE_EXT

The following USBD ioctl operations are exposed to user threads:

Operation	Description
USBD_REGISTER_MULTI_HC	Registers all the USB host controllers with USB system driver.
USBD_REGISTER_SINGLE_HC	Registers only a single USB host controller with USB system driver.
USBD_ENUMERATE_DEVICE	Gets a list of USB logical devices (excluding hubs) that are connected to a host controller.
USBD_ENUMERATE_ALL	Gets a list of all the USB logical devices that are connected to a host controller.
USBD_ENUMERATE_CFG	Gets a list of USB logical devices that are connected to a host controller along with the client device selection information.
USBD_GET_DESCRIPTORS	Gets standard USB descriptors for a logical device.
USBD_CFG_CLIENT_UPDATE	Updates client connection information.

Summary of USBD error conditions

Possible *errno* values for the adapter device driver are as follows:

Value	Description
EACCES	An openx subroutine was attempted to run while the adapter had one or more devices in use.
EEXIST	The device is already configured.
EINVAL	An invalid parameter or that the device is not opened.
EIO	• The command failed due to an error detected. • The device driver was unable to pin code. • A kernel service failed or an unrecoverable I/O error occurred.
ENOCONNECT	A USB bus fault occurred.
ENODEV	The target device cannot be selected or is not responding.
ENOMEM	The command cannot be completed because of an insufficient amount of memory.
ENXIO	The requested ioctl operation is not supported by this adapter.
EPERM	The caller does not have the required authority.

Related reference:

"Extensible Host Controller Adapter Device Driver" on page 236

"Enhanced Host Controller Adapter Device Driver" on page 240

"Open Host Controller Adapter Device Driver" on page 242

Related information:

usb0 special file

USBD ioctl Operations

There is a set of input and output control (ioctl) system calls to control I/O operations for Universal Serial Bus (USB) devices.

An `ioctl` call contains the following parameters:

- An open file descriptor.
- A request code.
- An integer value, possibly unsigned that is assigned to the driver.
- A pointer to data that is available in the host controller structure.

USBD_OPEN_DEVICE:

Purpose

Opens a specific Universal Serial Bus (USB) logical device.

Syntax

```
int fp_ioctl (file, USBD_OPEN_DEVICE, arg, ext)
```

Parameters

Item	Description
file	File descriptor that is obtained when the USB system driver (USBD) special file was opened.
arg	Address of an initialized `DEVOPEN` structure.
ext	Not used and must be set to zero.

Description

The client driver uses this `fp_ioctl` operation to establish a connection to a specific USB logical device that is identified by the information within the `DEVSELECTOR` structure. A USB logical device can be opened by only one client driver at a time. If a client opens the device, it must connect to a pipe before the data can flow to or from the device. This data includes, but is not limited to, the default control pipe. The client driver must close any device that it opened by calling the `usbdCloseDevice` operation when it no longer wants to manage the device. Typically, a client driver must open the USBD, issue a `USBD_OPEN_DEVICE ioctl` operation to open a specific USB device, and close the USBD. Then, the client must communicate with the USBD by using the handle that is returned by the `USBD_OPEN_DEVICE ioctl` operation and the interface macros that are located within the `usbdi.h` file. To properly track the USB device when the device needs to be moved or replaced, the client must open the device when the client is configured, and the client must close the USB device when the client is unconfigured.

Execution environment

This function can be called from the kernel process environment only.

Return values

Value	Description
0	Success.
Nonzero values	Failure.

USBD_OPEN_DEVICE_EXT:

Purpose

Opens a specific Universal Serial Bus (USB) logical device.

Syntax

```
int fp_ioctl (file, USBD_OPEN_DEVICE_EXT, arg, ext)
```

Parameters

Item	Description
file	File descriptor that is obtained when the USBD special file was opened.
arg	Address of an initialized DEVOPEN structure.
ext	Not used and must be set to zero.

Description

The client driver uses this fp_ioctl operation to establish a connection to a USB logical device that is identified by the information within the DEVSELECTOR structure. The ioctl operation is similar to the USBD_OPEN_DEVICE ioctl operation except that a client handle is allocated even when a USB logical device that matches the criteria that is specified in the DEVSELECTOR structure is not available. The USB system driver (USBD) returns the EAGAIN error value to indicate this condition. When the EAGAIN value is returned, the client driver must treat the device as disconnected and wait for connection before it proceeds with device initialization.

Execution environment

This function can be called from the kernel process environment only.

Return values

Value	Description
0	Success.
EAGAIN	No device matched the criteria. The client handle is valid but the device is treated as being in the disconnected state.
All other values	Failure.

USBD_REGISTER_MULTI_HC:

Purpose

Registers the Universal Serial Bus (USB) host controller with the USB system driver (USBD).

Syntax

```
int ioctl (file, USBD_REGISTER_MULTI_HC, arg)
```

Parameters

Item	Description
file	File descriptor that is obtained when the USBD special file was opened.
arg	Pointer to the information structure of the USB host controller.

Description

This `ioctl` operation registers all the USB host controllers that are listed in the `usb_adapterhc_info` structure with the USBD and allows the clients to communicate to the devices that are connected to the controller. There is no specific `ioctl` operation to unregister a hardware controller. It stays registered until either the USBD is unconfigured or the host controller is unconfigured. In the latter case, the host controller driver requests the USBD to unregister the host controller through the `usbdReqHCunregister` call vector. This `ioctl` operation must be invoked only by the `cfgusb` configuration method during enumeration and individual USB adapter configuration methods must use the `USBD_REGISTER_SINGLE_HC` operation to register single host controller instance.

Execution environment

This function can be called from the user process environment only.

Return values

Value	Description
0	Success.
-1	Failure. Check the `errno` value for specific failure causes.

USBD_REGISTER_SINGLE_HC:

Purpose

Registers single Universal Serial Bus (USB) host controller with the USB system driver (USBD).

Syntax

```
int ioctl (file, USBD_REGISTER_SINGLE_HC, arg)
```

Parameters

Item	Description
file	File descriptor that is obtained when the USBD special file was opened.
arg	Pointer to the integer that contains 32-bit `devno` structure of the USB host controller.

Description

This `ioctl` operation registers the specified host controller with the USBD and allows clients to communicate to the devices that are connected to the controller. There is no specific `ioctl` operation to unregister a hardware controller. It stays registered until either the USBD is unconfigured or the host controller is unconfigured. In the latter case, the host controller driver requests the USBD to unregister the host controller through the `usbdReqHCunregister` call vector.

Execution environment

This function can be called from the user process environment only.

Return values

Value	Description
0	Success.
-1	Failure. Check the errno value for specific failure causes.

USBD_ENUMERATE_DEVICE:

Purpose

Gets a list of USB logical devices (excluding hubs) that are connected to a host controller.

Syntax

```
int ioctl (file, USBD_ENUMERATE_DEVICE, arg)
```

Parameters

Item	Description
file	File descriptor that is obtained when the USBD special file was opened.
arg	Address of the USBENUM structure that is aligned on a 4-byte boundary.

Description

This ioctl operation returns a description of each logical USB device that is connected to the specified host controllers without any hubs. The description is returned in the form of a usb_device_t structure. The array of returned structures is encapsulated within a USBENUM structure whose length is specified by the caller. When this function is started, the devno and buffSize fields within the USBENUM structure must be initialized. The devno field must contain the 32-bit devno value of the host controller to be enumerated while the buffSize field must indicate the number of bytes that are available to buffer the returned array of usb_device_t structures. If the area is too small, the number of returned structures is truncated to fit the available space. The caller can detect this condition by noting that the number of returned usb_device_t structures is less than the number of discovered logical devices.

Execution environment

This function can be called from the user process environment only.

Return values

Value	Description
0	Success.
-1	Failure. Check the errno value for specific failure causes.

USBD_ENUMERATE_ALL:

Purpose

Gets a list of all the Universal Serial Bus (USB) logical devices that are connected to a host controller.

Syntax

```
int ioctl (file, USBD_ENUMERATE_ALL, arg)
```

Parameters

Item	Description
file	File descriptor that is obtained when the USB system driver (USBD) special file was opened.
arg	Address of the USBENUM structure that is aligned with a 4-byte boundary.

Description

This ioctl operation behaves in the same manner as the USBD_ENUMERATE_DEVICE ioctl operation except that it includes all hubs other than the root hub.

Execution environment

This function can be called from the user process environment only.

Return values

Value	Description
0	Success.
-1	Failure. Check the errno value for specific failure causes.

USBD_ENUMERATE_CFG:
Purpose

Gets a list of Universal Serial Bus (USB) logical devices that are connected to a host controller.

Note: This ioctl operation is used only by the USB system device driver's configuration method.

Syntax

```
int ioctl (file, USBD_ENUMERATE_CFG, arg)
```

Parameters

Item	Description
file	File descriptor that is obtained when the USB system driver (USBD) special file was opened.
arg	Address of the USBENUMCFG structure that is aligned with a 4-byte boundary.

Description

This ioctl operation behaves in the same manner as the USBD_ENUMERATE_DEVICE ioctl operation except that it also returns the client device selection information. The selection information uniquely identifies device-client pairing and allows the configuration method to correlate enumerated devices with their Object Data Manager (ODM) instances.

Execution environment

This function can be called from the user process environment only.

Return values

Value	Description
0	Success.
-1	Failure. Check the errno value for specific failure causes.

USBD_GET_DESCRIPTORS:

Purpose

Gets standard Universal Serial Bus (USB) descriptors for a logical device.

Syntax

```
int ioctl (file, USBD_GET_DESCRIPTORS, arg)
```

Parameters

Item	Description
file	File descriptor that is obtained when the USB system driver (USBD) special file was opened.
arg	Address of the USBDGD structure that is aligned on a 4-byte boundary.

Description

After a successful return from the ioctl operation, a DESCIDX structure is placed at the start of the specified buffer that is followed by the standard device descriptor, configuration descriptor, interface descriptor, endpoint descriptors, human interface device (HID) descriptor (if an HID device is used), hub descriptor (if hub device is used), and string descriptors of the specified logical USB device. The DESCIDX structure provides direct addressability to the individual descriptors. String descriptors are reformed to null terminated American Standard Code for Information Interchange (ASCII) strings for ease of use. All other descriptors adhere to the standard USB format. Since the size of the returned data is typically unknown, the ioctl operation must be called twice. The first time that you call the ioctl operation, set the bufferLength field equal to zero and the buffer field to null. The ioctl operation might fail with the ENOSPC error, however the minBuffLength value is returned that indicates the required size of the buffer. The caller can then allocate the buffer and call the ioctl operation again with the bufferLength field set to the correct value.

Execution environment

This function can be called from the user process environment only.

Return values

Value	Description
0	Success.
-1	Failure. Check the errno value for specific failure causes.

USBD_CFG_CLIENT_UPDATE:

Purpose

Updates the client connection information.

Note: This ioctl operation is used only by the USB system driver (USBD) configuration method.

Syntax

```
int ioctl (file, USBD_CFG_CLIENT_UPDATE, arg)
```

Parameters

Item	Description
file	File descriptor that is obtained when the USBD special file was opened.
arg	Address of the `USBENUMCFG` structure that is aligned with a 4-byte boundary.

Description

This `ioctl` operation is used by the USBD's defined children configuration procedure to update the device selection criteria that is used by the client driver. Specifically, it updates the `hcdevno`, `addr`, `cfg`, and `intfc` fields to reflect the current values for the device that are managed by the client.

Execution environment

This function can be called from the user process environment only.

Return values

Value	Description
0	Success.
-1	Failure. Check the `errno` value for specific failure causes.

USBLIBDD Passthru Driver

Purpose

Supports the application drivers that are written by using the **libusb** APIs.

Syntax

```
#include <usbdi.h>
#include <usb.h>
```

Description

The **libusb passthru** driver is the layer between the user-level application driver and the USB protocol driver (USBD). The **/dev/usblibdevX** special file provides interface to **libusb** applications to communicate directly with the device through the **passthru** driver. The **passthru** driver converts the **libusb** APIs to the USBD function vectors that interact with the appropriate Universal Serial Bus (USB) host controllers such as Open Host Controller Interface (OHCI), Enhanced Host Controller Interface (EHCI), or eXtensible Host Controller Interface (xHCI).

The **libusb** devices are created in the **/dev** file system irrespective of the presence of built-in AIX client USB drivers. The parent for this device is the **usb0** and the **libusb** devices that are enumerated by the **/usr/lib/methods/cfgusb** file.

The **/usr/lib/drivers/usb/usblibdd** driver implements the **libusb passthru** driver. The **passthru** driver uses the **/usr/lib/methods/cfgusblibke** configuration method. These devices have two Object Data Manager (ODM) attributes called **usbdevice** and **speed**. If a device belongs to standard classes such as Mass Storage, Tape, human interface device (HID), these devices are claimed by the built-in USB client driver of the AIX operating system. In such case, a **libusb** device is created as a pseudo device. The **usbdevice** attribute identifies the device of the client driver that is associated with a pseudo device. If a device belongs to other classes and if client drivers are not associated with the device, the **usbdevice** attribute is not valid.

For every **libusb** device, which has an AIX operating system built-in client driver, a new attribute that is called **usbdevice** is created in the ODM to identify the client driver device that is associated with the **libusb** device. The following example shows how the device is displayed:

```
# lsattr -El usblibdev0

speed    highspeed  USB Protocol Speed of Device      False

usbdevice usbms0 Actual USB Device with Client Driver False
```

In this example, the USB device that is connected is a flash drive, which has the AIX operating system built-in **/usr/lib/drivers/usb/usbcd** Mass Storage Class client driver. The device of the client driver associated with the **usblibdev0** device is **usbms0**.

By default, this device is claimed by the built-in client driver. The same device is also claimed by the **libusb passthru** driver. Therefore, for one physical USB device, you have two OS devices (**usbms0** and **usblibdev0**) located in the **/dev** file system after you run the configuration method of the parent device, which is USBD protocol driver.

Note: Only the built-in client driver or the **libusb passthru** driver can use this device at a time. You cannot run simultaneous operations on both drivers.

Use the following command to display the USB devices in this scenario:

```
# lsdev -C | grep usb
```

An output similar to the following example is displayed:

```
usb0       Available       USB System Software

usbhc0     Available 00-00 PCIe2 USB 3.0 xHCI 4-Port Adapter (4c10418214109e04)

usblibdev0 Available       USB Library Interface Device

usbms0     Available 0.3   USB Mass Storage
```

In the this example, a USB encryption device (vendor-defined class) is connected to the AIX system. The device does not have a built-in client driver. This device is only claimed by the **libusb passthru** driver and only a single device is displayed. Another example to display the USB devices follows:

```
# lsdev -C | grep usb
```

An output similar to the following example is displayed:

```
usb0       Available       USB System Software

usbhc0     Available 00-00 PCIe2 USB 3.0 xHCI 4-Port Adapter (4c10418214109e04)

usblibdev1 Available 0.4   USB Library Interface Device
```

In this example, **usblibdev1** device is the encryption device of the **libusb** driver.

Device-dependant subroutines

The **libusb passthru** driver supports the following subroutines:

- open
- close
- ioctl

Note: The **read** and **write** subroutines are not supported.

open and close subroutines

The **open** and **close** subroutines are not directly supported on **usblibdevX** devices. You can **open** and **close** subroutines by using the **libusb** APIs.

ioctl subroutine

The **libusb** driver exposes the **ioctl** subroutine to the **libusb** user environments. The **libusb** implementation of operating system backend use these **ioctl** subroutines. The **ioctl** subroutine acts as a pass through between the application and the protocol driver.

The following USBD `ioctl` operations are supported by the **libusb** drivers:

Table 2. USBD ioctl operations

Operations	**Description**
USBLIB_PIPE_IO	Issues I/O on the wanted endpoint through the **aix_pipeio** structure.
USBLIB_HALT_CLEAR	Issues a request to halt an endpoint.
USBLIB_GETIRP_STATUS	Read the status of the I/O request packet (IRP) that was issued.
USBLIB_SET_CONFIGURATION	Issues a set configuration on a device.
USBLIB_CLAIM_INTERFACE	Ensures sure that the interface is being used by the **libusb passthru** driver.
USBLIB_RELEASE_INTERFACE	Sets the interface to alternate setting value of zero.
USBLIB_SET_ALT_INTFC	Sets the alternate setting on an interface.
USBLIB_RESET_DEVICE	Resets on the device.
USBLIB_ABORT_IO	Aborts or cancels to the submitted I/O.
USBLIB_GET_CONFIG_DESC	Issues request to read the entire configuration descriptor. If a device has X configurations, X number of total configuration descriptor is read and stored in a single buffer.

USBD error conditions

Possible *errno* values for the adapter device driver follow:

Table 3. USBD error conditions

Value	**Description**
EAGAIN	Indicates that the operation has been to retried.
EEXIST	The device is already configured.
EINVAL	An invalid parameter or the device is not opened.
EIO	• The command failed due to an error. • The device driver was unable to pin code. • A kernel service failed or an unrecoverable I/O error occurred.
ENOCONNECT	A USB bus fault occurred.
ENODEV	The target device cannot be selected or is not responding.
ENOMEM	The command cannot be completed because of insufficient memory.
ENXIO	The requested **ioctl** operation is not supported by this adapter.
EPERM	The caller does not have the required authority.

Non-responsive USB devices

USB devices that are associated to libusb applications on the AIX operating system might not respond on non-control endpoints. This condition might be because of default behavior of AIX USB protocol driver to send the CLEAR_FEATURE request when the USB devices are opened.

Note: The CLEAR_FEATURE request is a standard USB command to remove the halt condition on the device.
To resolve the issues associated with the non-responsive USB devices, the following attributes must be set:

PdAt class object
The Predefined Attribute (PdAt), object class contains an entry for each existing attribute for each USB device. This includes information such as interrupt levels, bus I/O address ranges, baud rates, parity settings, block sizes, and microcode file names. To initialize the PdAt class object set the following values:

```
uniquetype = "generic/usbif/usblibke"
attribute = "<vendorid>_<productid>"
deflt = "toggle_no"
values = "toggle_yes,toggle_no"
width = ""
type = "R"
generic = "U"
rep = "sl"
nls_index = 0
```

VendorId
Vendor ID of the USB device that can be obtained from the descriptor data of the USB device. VendorID must be a hexadecimal number.

ProductID
Product ID of the USB device that can be obtained from the descriptor data of the USB device. ProductID must be a hexadecimal number.

toggle_no
Indicates that the CLEAR_FEATURE request is not sent to the device during pipe initialization.

toggle_yes
Indicates the default behavior of the USB device to send the CLEAR_FEATURE request.

Note: The ODM entry must be added to each device that does not respond on non-control endpoints.

An example ODM attribute follows. This example considers the Kingston USB flash drive (DataTraveler Ultimate G2). The vendor ID of Kingston USB flash drive is 0951 and Product ID is 1656.

```
PdAt:
        uniquetype = "generic/usbif/usblibke"
        attribute = "0951_1656"
        deflt = "toggle_no"
        values = "toggle_yes,toggle_no"
        width = ""
        type = "R"
        generic = "U"
        rep = "sl"
        nls_index = 0
```

To add predefined attributes to the PdAt object class complete the following steps:

1. Run the following command to remove a non-responsive USB device from the ODM entries.

   ```
   rmdev -Rl usb0
   ```

   The output might be similar to the following example:

```
usbms0 Defined
usblibdev0 Defined
usblibdev1 Defined
usb0 Defined
```

2. Add the `odm PdAt` entry in to a file by using any standard file edit command such as `vi`.
3. Run the following command.

   ```
   odmadd entry
   ```

   The output is not displayed.
4. Run the following command.

   ```
   cfgmgr -l usb0
   ```

   The output is not displayed.
5. Run the `libusb` application. Following example shows the execution of a USB application.

   ```
   # ./xusb -k XXXX:YYYY
   Opening device XXXX:YYYY...
   found /dev/usbhc0
   found 1 devices
   found /dev/usbhc1
   found 0 devices
   found /dev/usbhc2
   found 1 devices

   Reading device descriptor:
               length: 18
         device class: 0
                  S/N: 0
              VID:PID: XXXX:YYYY
            bcdDevice: 0303
      iMan:iProd:iSer: 1:2:0
             nb confs: 1

   Reading BOS descriptor: no descriptor

   Reading first configuration descriptor:
                nb interfaces: 1
                 interface[0]: id = 0
   interface[0].altsetting[0]: num endpoints = 1
      Class.SubClass.Protocol: 03.00.00
          endpoint[0].address: 81
              max packet size: 0008
             polling interval: 0A

   Claiming interface 0...

    in aix_claim_interface

   Reading string descriptors:
      String (0x01): "DeviceName"
      String (0x02): "Elitename"

   Releasing interface 0...
   Closing device...
   ```
6. Run the following command to delete the `PdAt` entry of the non-responsive USB device:

   ```
   odmdelete -o PdAt -q 'attribute=0951_1656 and uniquetype="generic/usbif/usblibke"'
   ```

   The output might be similar to the following example:

   ```
   0518-307 odmdelete: 1 object deleted
   ```

Related information:

Enabling/Disabling LIBUSB Devices

Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year).

Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

Special characters

/dev/nvram special file
- machine device driver and 24

A

adapter cards
- device method guidelines for 16

adapters
- bus resources 36
- PdAt object class
  - considerations 4

attrval subroutine 5
autodial protocols 71

B

bus resources
- allocating 6

bus special file
- machine device driver 24

busresolve subroutine 6

C

CD-ROM SCSI device driver 151
cfg device method 47
CFG_INIT operation
- PCI MPQP 86

CFG_TERM operation
- PCI MPQP 86

Change method 45
- handling invalid attributes 45

chg device method 45
CIO_GET_FASTWRT operation
- ddioctl 58

CIO_GET_STAT operation
- ddioctl 62
- PCI MPQP 59

CIO_HALT operation
- ddioctl 65
- PCI MPQP 64

CIO_QUERY operation
- ddioctl 68
- PCI MPQP 66

CIO_START operation
- ddioctl 73
- PCI MPQP 69

close subroutine
- /dev/bus special file 24
- /dev/nvram special file and 24
- rmt SCSI device driver and 215
- scdisk SCSI device driver and 151
- SCSI adapter device driver and 143
- tmscsi SCSI device driver and 227

communication I/O subsystem 65
communications device handlers 84, 93
- checking event status 81
- communications device handlers 93

communications device handlers *(continued)*
- communications sessions
  - halting 65
  - opening 73
- device statistics
  - returning 68
- entry points
  - dd_fastwrt 74
  - ddclose 75
  - ddopen (kernel mode) 76
  - ddopen (user mode) 78
  - ddread 79
  - ddselect 81
  - ddwrite 82
- fast-write call 58
- kopen_ext parameter block 76
- query_parms parameter block 68
- queuing messages 82
- reading data messages 79
- session_blk parameter block 73
- status blocks
  - getting 62
- system resources
  - freeing 75
- transmitting data 74
  - PCI MPQP 93

Config_Rules object class 8
Configuration Manager
- rules
  - configuration 8

Configure method
- and errors 47
- and VPD 47
- described 47
- guidelines 47

CuAt object class
- attribute information
  - updating 42
- creating objects 42
- deleting objects 42
- described 9
- descriptors 9
- getattr subroutine 20
- putattr subroutine 42
- querying attributes 20

CuDep object class
- descriptors 11
- introduction 11

CuDv object class
- descriptors 12
- generating logical names 19
- genminor subroutine 17
- subroutines
  - genseq 19

CuDvDr object class
- descriptors 11
- genmajor subroutine 16
- getminor subroutine 21
- major numbers
  - releasing 43, 44

CuDvDr object class *(continued)*
minor numbers
releasing 43
querying minor numbers 21
reldevno subroutine 43
relmajor subroutine 44
CuVPD object class
descriptors 15
introduction 15

D

data messages
reading 79
data structures
allocating
for communications PDH 78
initializing
for communications PDH 76, 78
dd_fastwrt entry point 74
ddclose entry point 75
ddread entry point
communications PDH 79
ddselect entry point
communications PDH 81
ddwrite entry point
communications PDH 82
def device method 51
Define method 51
device attributes
creating 42
deleting 42
predefined 32
querying class 20
specific 9
updating 42
verifying ranges 5
device configuration methods
guidelines for writing 57
device configuration subroutines
attrval 5
busresolve 6
genmajor 16
genminor 17
genseq 19
getattr 20
getminor 21
loadext 23
putattr 42
reldevno 43
relmajor 44
device driver
loading 24
machine
/dev/bus special file 24
/dev/nvram special file 24, 30
bus special file 24
initialization 24
overview 24
termination 24
major numbers
generating 16
names
obtaining 23
device drivers
sctape FC 209
device methods
adapter card guidelines 16
Change 45
Configure 47
Define 51
returning errors 22
Start 57
Stop 57
Undefine 56
devices
critical resource information
storing 11
defined state
resolving attributes of 6
dependencies 11
generating minor numbers 17
intermediate
connection information 38
logical names
generating 19
major numbers
releasing 43
minor numbers
releasing 43
types of 38

E

EHCI 240
Enhanced Host Controller Interface
adapter device driver 240
Ethernet device handler
write_extension parameter block 83
Extensible Host Controller Interface
adapter device driver 236

G

genmajor subroutine 16
genminor subroutine 17
genseq subroutine 19
getattr subroutine 20
getminor subroutine 21

H

HCD_REGISTER_HC
ioctl operation 245
HCD_REQUEST_COMPANIONS
ioctl operation 242

I

idscsi 226, 227
intermediate devices
connection information 38
IOCINFO operation
tmscsi 142
ioctl operations
/dev/nvram special file 24
ioctl subroutine
rmt SCSI device driver and 215
scdisk SCSI device driver and 151
SCSI adapter device driver and 143
tmscsi SCSI device driver and 227

ioctl subroutines
  /dev/bus special file 24
  /dev/nvram special file 24

K

kernel extensions
  loading 23
  unloading 23
kopen_ext parameter block 76
KSTDELAY 107

L

loadext subroutine 23
logical names 19

M

machine device drivers 24
magnetic tape access
  tape SCSI device driver and 215
major numbers
  generating 16
  releasing 43, 44
message queues
  messages
    queueing for transmission 82
microcode
  downloading to SCSI adapter 172
minor numbers
  generating 17
  getting 21
  releasing 43
MP_CHG_PARMS operation 84
MREGRINGEXT 114

N

NVMe
  controller
    Device Driver 235
  storage (hdisk)
    Device Driver 234
NVMe controller Device Driver 235
NVMe storage (hdisk) Device Driver 234

O

ODM
  object classes 31
OHCI 242
Open Host Controller Interface
  adapter device driver 242
open subroutine
  /dev/bus special file 24
  /dev/nvram special file and 24
  rmt SCSI device driver and 215
  scdisk SCSI device driver and 151
  SCSI adapter device driver and 143
  tmscsi SCSI device driver and 227

P

passthru subroutine 128
PCI MPQP device handler
  allocating channels 88
  controlling 86
  deallocating channels 88
  entry points
    tsclose 84
    tsconfig 86
    tsioctl 86
    tsmpx 88
    tsopen 89
    tsread 90
    tsselect 92
    tswrite 93
  events
    checking for 92
  getting status of 59
  initializing 86
  ioctl operations
    CIO_GET_STAT 59
    CIO_HALT 64
    CIO_QUERY 66
    CIO_START 69
    MP_CHG_PARMS 84
  opening for transmission 89
  read_extension parameter block 90
  reading data 90
  resetting 84
  sessions
    ending 64
    starting 69
  t_err_threshold structure 72
  t_start_dev structure 70
  terminating 86
  tswrite parameter block 94
  using autodial protocols 71
PdAt object class
  attrval subroutine 5
  descriptors 32, 36
  getattr subroutine 20
  loading devices 24
  querying attributes 20
  types of attributes 32
PdCn object class 38
PdDv object class
  adapter-specific considerations 4
  descriptors 38
  loadext subroutine 23
  loading devices 24
piocmdout subroutine 129
pioexit subroutine 130
piogetopt subroutine 132
piogetstr subroutine 134
piogetvals subroutine 135
piomsgout subroutine 137
predefined attributes 32
print formatters
  attribute database
    initializing 135
  attribute variables 135
  attributes
    retrieving 134
  command-line flags
    parsing 132
    processing 141
  converting attribute strings 135

print formatters *(continued)*
- database
  - validating input parameters 141
- database variables
  - initializing 141
- exiting 130
- flag arguments
  - converting 132
  - overlaying defaults 132
- passing input data stream 128
- sending messages from 137
- subroutines
  - list for writing 142
  - list of 142
  - passthru 128
  - piocmdout 129
  - pioexit 130
  - piogetopt 132
  - piogetstr 134
  - piogetvals 135
  - piomsgout 137
  - restore 140
  - setup 141

printer attribute variables 135
putattr subroutine 42

Q

query_parms parameter block 68

R

read subroutine
- /dev/bus special file 24
- /dev/nvram special file 24
- tmscsi SCSI device driver and 227

read_extension parameter block 90
readx subroutine
- scdisk SCSI device driver and 151

reldevno subroutine 43
remajor subroutine 44
restore subroutine 140
rmt SCSI device driver
- close subroutine and 215
- device-dependent subroutines 215
- error conditions 215
- error record values 215
- ioctl subroutine and 215
- open subroutine and 215
- reliability and serviceability 215

S

scdisk SCSI device driver
- close subroutine and 151
- device requirements 151
- device-dependent subroutines 151
- error conditions 151
- error record values 151
- ioctl subroutine and 151
- open subroutine and 151
- physical volume and CD-ROM 151
- readx subroutine and 151
- reliability and serviceability 151
- writex subroutine and 151

SCIOCMD operation 169
SCIODIAG operation 171
SCIODNLD operation 172
SCIOEVENT operation 173
SCIOGTHW operation 175
SCIOHALT operation 175
SCIOINQU operation 176
SCIOREAD operation 177
SCIORESET operation 179
SCIOSTART operation 180
SCIOSTARTTGT operation 181
SCIOSTOP operation 183
SCIOSTOPTGT operation 184
SCIOSTUNIT operation 185
SCIOTRAM operation 186
SCIOTUR operation 187
scsesdd SCSI Device Driver 188
SCSI adapter device driver 143, 227
- close subroutine and 143
- closing logical paths 183, 184
- device registration 173
- device-dependent subroutines 143
- downloading microcode 172
- error conditions 143
- error-record values 143
- halting a device 175
- ioctl subroutine and 143
- issuing commands 169
- issuing diagnostic commands 171
- issuing inquiry commands 176
- issuing read command 177
- managing dumps 143
- open subroutine and 143
- opening logical paths 180, 181
- reliability and serviceability 143
- resetting a device 179
- starting devices 185
- supporting the SCSI adapter 143
- testing a unit 187
- testing buffer RAM 186
- testing card DMA interface 186
- verifying gathered write support 175

SCSI ioctl operations
- SCIOCMD 169
- SCIODIAG 171
- SCIODNLD 172
- SCIOEVENT 173
- SCIOGTHW 175
- SCIOHALT 175
- SCIOINQU 176
- SCIOREAD 177
- SCIORESET 179
- SCIOSTART 180
- SCIOSTARTTGT 181
- SCIOSTOP 183
- SCIOSTOPTGT 184
- SCIOSTUNIT 185
- SCIOTRAM 186
- SCIOTUR 187

SCSI subsystem 173
sctape FC device driver 209
select entry point
- tmscsi SCSI device driver and 227

session_blk parameter block 66, 73
setup subroutine 141
Start method 57
status blocks
- getting 62

Stop method 57
stp device method 57
stt device method 57

T

t_err_threshold structure 72
t_start_dev structure 70
tape device media errors 215
tape SCSI device driver
  introduced 215
TIMIORESET operation 226
TMCHGIMPARM operation 221
TMGETSENS operation 222
TMIOASYNC operation 223
TMIOCMD operation 223
TMIOEVNT operation 224
TMIOSTAT operation 226
tmscsi ioctl operations
  IOCINFO 142
  TMCHGIMPARM 221
  TMGETSENS 222
  TMIOASYNC 223
  TMIOCMD 223
  TMIOEVNT 224
  TMIORESET 226
  TMIOSTAT 226
tmscsi SCSI device driver
  changing parameters 221
  close subroutine and 227
  configuring 227
  device-dependent subroutines 227
  error logging 227
  getting device information 142
  getting device status 226
  ioctl subroutine and 227
  open subroutine and 227
  processor-to-processor communications 227
  querying event status 224
  read subroutine and 227
  requesting sense data 222
  select entry point and 227
  sending bus device resets 226
  sending direct commands 223
  transferring data asynchronously 223
  write subroutine and 227
tsclose entry point 84
tsconfig entry point 86
tsioctl entry point 86
tsmpx entry point 88
tsopen entry point 89
tsread entry point 90
tsselect entry point 92
tswrite entry point 93
tswrite parameter block 94

U

udef device method 56
Unconfigure method 54
Undefine method 56
USB audio
  device driver 245
USB keyboard
  device driver 246
USB mass storage
  device driver 248
  error conditions 258
USB mouse
  device driver 264
USB subsystem 236
USB tape
  device driver 265
  error conditions 269
USBD ioctl operation
  USBD_CFG_CLIENT_UPDATE 283
  USBD_ENUMERATE_ALL 281
  USBD_ENUMERATE_CFG 282
  USBD_GET_DESCRIPTORS 283
  USBD_OPEN_DEVICE 278
  USBD_OPEN_DEVICE_EXT 279
  USBD_REGISTER_MULTI_HC 279
  USBD_REGISTER_SINGLE_HC 280
USBD IOCTL operation
  USBD_ENUMERATE_DEVICE 281
USBD ioctl operations 278
USBD protocol driver 276, 284

V

vital product data 47
VPD 15
  handling 47

W

write subroutine
  /dev/bus special file 24
  /dev/nvram special file 24
  tmscsi SCSI device driver and 227
write_extension parameter block 83
writex subroutine
  scdisk SCSI device driver and 151

X

XHCI 236

Printed in USA