NAME

`intro` – Introduction to special files

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The entries in this section describe the characteristics of the device interfaces (device drivers) and corresponding hardware devices or pseudo devices in UNICOS; there is one entry or set of related entries per page.

UNICOS devices and pseudo devices are represented by special files in the `/dev` directory. With a few exceptions, each hardware device is represented by one or more files in `/dev`. Examples of devices are disk drives, which are represented by the special files in the `/dev/dsk` directory and are described by the `dsk`(4) entry. *Pseudo devices* are device drivers that have no associated hardware but which behave in much the same way as a hardware device. Examples of pseudo devices are the null pseudo device, `/dev/null` (described by the `null`(4) entry), and the pseudo terminals, located in the `/dev/pty` directory (described by the `pty`(4) entry).

Three types of special files exist: block special files, character special files, and FIFO special files (named pipes). This manual does not discuss FIFO special files; for information on these files, see `pipe`(2).

On *block devices*, data read from or written to the device is moved through a cache of system buffers. In contrast, devices that do not use the system buffer cache are *character devices*. (Character devices do not necessarily move data 1 character at a time; in fact, very large blocks (track-sized or cylinder-sized) are often used.) The unbuffered I/O of character devices is often called *raw* I/O mode.

On CRAY Y-MP systems, disks and RAM disks are the only block devices supported. These devices are documented in `dsk`(4) and `ram`(4). The supported block devices are disks, buffer memory resident (BMR) file systems, the SSD solid-state storage device, and RAM disks. The `dsk`(4) and `ram`(4) entries describe these devices.

You can use disk drives as character devices instead of block devices; in this case, the system buffer cache is not used, and the data is moved directly between the device and the user's buffer. The `fcntl`(2) system call is available to open the block special file in raw mode (see `fcntl`(5) and `dsk`(4)).

Most devices and pseudo devices can do read and write operations; many can do additional operations. The capabilities of each device are discussed in the entry for that device.

Many devices allow further manipulation of the device through the special file with the `ioctl`(2) system call. For example, a process can issue an `ioctl` request to return the status of a device; the `ioctl` request `CPUSTAT` returns the status of the target CPU (see `cpu`(4)). The `ioctl` requests are device-dependent, and they are discussed in the entry for each device if appropriate.

Special files are created using the `mknod`(8) command, which builds a directory entry and an inode for a device. For specific information on creating devices, see the `mknod`(8) command and the system installation bulletin for your UNICOS release.

SEE ALSO

`cpu`(4), `fcntl`(2), `ioctl`(2), `pipe`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`cpu` – Interface to special CPU functions

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `cpu` driver allows control of many different aspects of the CPU environment in which processes run. Some of the functions refer to a specific physical CPU. The naming convention for the special files corresponding to CPUs is as follows:

CPU 0	`/dev/cpu/0`
CPU 1	`/dev/cpu/1`
CPU 2	`/dev/cpu/2`
CPU 3	`/dev/cpu/3`
CPU 4	`/dev/cpu/4`
CPU 5	`/dev/cpu/5`
CPU 6	`/dev/cpu/`*ncpus*

The following usage, while still accepted, may not be supported in future releases.

CPU 0	`/dev/cpu/a`	CPU 8	`/dev/cpu/i`
CPU 1	`/dev/cpu/b`	CPU 9	`/dev/cpu/j`
CPU 2	`/dev/cpu/c`	CPU 10	`/dev/cpu/k`
CPU 3	`/dev/cpu/d`	CPU 11	`/dev/cpu/l`
CPU 4	`/dev/cpu/e`	CPU 12	`/dev/cpu/m`
CPU 5	`/dev/cpu/f`	CPU 13	`/dev/cpu/n`
CPU 6	`/dev/cpu/g`	CPU 14	`/dev/cpu/o`
CPU 7	`/dev/cpu/h`	CPU 15	`/dev/cpu/p`

Some of the functions refer to a CPU in which the specified process (or processes) may happen to execute. For those functions, you should use the special file `/dev/cpu/any`. This special file provides a generic interface to any CPU.

For sites with more than one CPU, create as many CPU devices as needed, changing the name and minor device number in sequence.

The only valid system calls to the `cpu` driver are `open`(2), `close`(2), and `ioctl`(2).

The available `ioctl` requests are defined in the `sys/cpu.h` include file.

For the following requests, *arg* is interpreted as a pointer to an exchange packet. The following `ioctl` requests are supported for a specific CPU:

`CPU_DOWN`	Disables the CPU.

CPU_START	Copies the exchange package to which *arg* points to the system diagnostic and stores it. The exchange information is assumed to be absolute address 0 of the stand-alone program. The target CPU must be in a down state. The base address (BA) in the exchange package is considered as relative to the caller's base address; the limit address (LA) is set to that of the caller. CPU_START is disabled for CRAY T90 series architecture. This ioctl request will return a value of EINVAL.
CPU_STAT	Copies the system's user exchange package to *arg*. CPU_STAT copies the user exchange information only if the target CPU exits.
CPU_DSTAT	Copies the system's diagnostic exchange package to *arg*. CPU_DSTAT copies the diagnostic exchange information only if the target CPU exits.
CPU_STOP	Halts the target CPU; *arg* is not used but must be supplied. The CPU must have been started by using PU_START. CPU_STOP is disabled for CRAY T90 series architecture. This ioctl request will return a value of EINVAL.
CPU_UP	Reenables the CPU. Permits the CPU to be scheduled for normal processing. The CPU must have been downed previously (CPU_DOWN).
CPU_UP_S_CACHE	Enables scalar cache for the specified CPU. (CRAY T90 and CRAY J90 series)
CPU_DN_S_CACHE	Disables scalar cache for the specified CPU. (CRAY T90 and CRAY J90 series)

The following ioctl requests apply only to CRAY Y-MP systems that are supported on the generic device (/dev/cpu/any). These requests require the following structure (except for the CPU_CLRTMR request). This structure is defined in the sys/cpu.h include file, as follows:

```
struct  cpudev  {
        int   cat;        /* category   */
        int   id;         /* identifier */
        long  word;       /* parameter  */
        long  cpu_word1;  /* parameter  */
        long  cpu_word2;  /* parameter  */
        long  cpu_word3;  /* parameter  */
        long  cpu_word4;  /* parameter  */
};
```

CPU_CLRRT	Clears real-time status from the specified process or process group.
CPU_CLRTMR	Removes the calling process from the interval timer queue. The cpudev structure is not required.

`CPU_CLUSTER` Selects the clusters specified by the bit mask in `word` for the ID and category specified by `id` and `cat`. If `id` is 0, the current process ID is assumed. Bit 2^k in the mask refers to cluster *k*. The resulting bit mask is returned. Clusters 0 and 1 cannot be selected.

`CPU_DEDICATE` Dedicates the CPUs specified by the bit mask in `word` to the ID and category specified by `id`. If `id` is 0, the current process ID is assumed. Bit 2^k in the mask refers to CPU *k*. If the mask contains 0 bits for CPUs that are already dedicated, those CPUs are released from dedication. The resulting bit mask is the return value of `CPU_DEDICATE`.

`CPU_GETMODE` Returns a bit mask of the current mode settings for a process in a process group. The bit positions in the mask (right-justified) are as follows:

`UXP_MON`	0	Monitor mode
`UXP_BDM`	1	Bidirectional memory
`UXP_EMA`	2	Extended mode addressing
`UXP_AVL`	3	Second vector logical
`UXP_IFP`	4	Interrupt on floating-point error
`UXP_IOR`	5	Interrupt on operand range error
`UXP_ICM`	6	Interrupt on correctable memory errors
`UXP_IUM`	7	Interrupt on uncorrectable memory errors
`UXP_IMM`	8	Interrupt monitor mode
`UXP_RPE`	9	Register parity interrupts enabled
`UXP_SCE`	10	Scalar cache enabled
`UXP_IIO`	11	I/O interrupts enabled
`UXP_IPC`	12	Programable clock interrupts enabled
`UXP_IAM`	13	AMI interrupts enabled
`UXP_IXI`	14	Exceptional input interrupt enabled (IEEE)
`UXP_INX`	15	Inexact interrupt enabled (IEEE)
`UXP_IUN`	16	Underflow interrupt enabled (IEEE)
`UXP_IOV`	17	Overflow interrupt enabled (IEEE)
`UXP_IDV`	18	Divide by zero interrupt enabled (IEEE)
`UXP_INV`	19	Invalid interrupt enabled (IEEE)
`UXP_RM0`	20	Round mode 0 set (IEEE)
`UXP_RM1`	21	Round mode 1 set (IEEE)

`CPU_QDOWN` Returns a mask of the CPUs down. The number of CPUs configured is returned in `word`. The mask of down CPUs is returned in `cpu_word1`.

`CPU_RTFRAME`	Enables least-time-to-go scheduling for the calling process. Requires the following parameters: `cpuctl.rtframe` Count of milliseconds in frame `cpuctl.rtitm` Count of milliseconds of input time `cpuctl.rtctm` Count of milliseconds of required computer time `cpuctl.rtotm` Count of milliseconds of required output time A total frame time in `os_hz` units is passed in `cpuctl.rtframe`. This value is used for least-time-to-go scheduling of real-time processes. An optional signal number may be passed in `word1`. If nonzero, this signal will be sent each time the frame time expires. You can send this command at any time to resynchronize the frame start point. The category must be `C_PROC`, the ID must be 0 or the caller's process ID, and the process must be real time.
`CPU_RTPERMIT`	Used by super-user processes to bestow permission to nonsuper-user processes; used by process groups to request real-time status.
`CPU_SELECT`	Selects the CPUs specified by the bit mask in `word` for the ID and category specified by `id` and `cat`. If `id` is 0, the current process ID is assumed. Bit 2^k in the mask refers to CPU *k*. The resulting bit mask is the return value of `CPU_SELECT`.
`CPU_SETMODE`	Sets mode bits. `word` is a bit mask. The value 1 in a bit position enables the mode; the value 0 disables the mode. The bit positions in the mask are the same as those in the `ioctl` request `CPU_GETMODE`. If the calling process is not a super-user process, `UXP_MON`, `UXP_IIO`, `UXP_IPC`, and `UXP_IAM` set to 1 returns the `EPERM` error.
`CPU_SETRT`	Marks the ID and category specified by `id` and `cat` as being real-time processes. Real-time processes are scheduled from a separate run queue that is always checked before the default run queue. Priorities for real-time processes are initialized as are other processes, but they are not adjusted by the system. The run queue can be ordered by a change in nice values (see `nice`(2)). Real-time processes have permissions that allow them to use clock and CPU dedication even though they may not be a super user. The `EPERM` error is returned if the calling process is not a super-user process or does not have permission (see `CPU_RTPERMIT`).
`CPU_SETTMR`	Places the calling process on the interval timer queue. The interval (in milliseconds) is in `word`. The process then receives `SIGALRM` signals at the specified interval.

CPU_GETMAXERR	Returns values from kernel maxerrint table. The maximum PRE count is returned in cpu_word1. The maximum ORE count is returned in cpu_word2. The maximum ERR count is returned in cpu_word3. The values in the maxerrint table specify the maximum count of Program Range Errors, Operand Range Errors, and Error Exits that are allowed for a process in one connection to a CPU. If any of the counts is exceeded, the process is sent a SIGKILL signal. A maxerrint table value of 0 means that no limit is enforced for that particular error.
CPU_SETMAXERR	Sets values into the kernel maxerrint table. word is a bit mask that indicates which fields should be set. If bit 2**0 of the mask is set, the maximum PRE count is set to the value in cpu_word1. If bit 2**1 of the mask is set, the maximum ORE count is set to the value in cpu_word2. If bit 2**2 of the mask is set, the maximum ERR count is set to the value in cpu_word3.

In addition to the standard ioctl error codes (see ioctl(2)), the following are errors that cause an ioctl request to fail:

EFAULT	Exchange information address (*arg*) is out of the user's memory area
EINVAL	Nonexistent CPU is requested, category is an unknown type, or timer interval is 0
ENODEV	Last CPU is being stopped by using CPUSTOP
ENOENT	Timer queue is full
EPERM	User is not super user or does not own the device for device-specific (CPU-specific) requests
ESRCH	No existing process matches the category and ID specified

FILES

/dev/cpu/0	Device driver for CPU 0
/dev/cpu/1	Device driver for CPU 1
/dev/cpu/2	Device driver for CPU 2
/dev/cpu/3	Device driver for CPU 3
/dev/cpu/4	Device driver for CPU 4
/dev/cpu/5	Device driver for CPU 5
.	
.	
.	
/dev/cpu/32	Device driver for CPU 32
/dev/cpu/any	Device driver for any CPU

`/usr/include/sys/category.h`	CPU `ioctl` request definitions
`/usr/include/sys/cpu.h`	

SEE ALSO

`close`(2), `ioctl`(2), `nice`(2), `open`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

disk – Physical disk interface

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The /dev/disk device is used as the interface to all of the real physical disk devices configured. The ioctl(2) system call is used to issue requests to individual devices by passing the ASCII name of the physical device in the control structure to the ddcntl.c device driver. The ASCII names of the physical devices are those configured in cf/conf.*SN*.c (*SN* is the mainframe serial number) or the I/O subsystem (IOS) parameter file.

The control structure used for the ioctl system call is defined in the sys/ddcntl.h include file, as follows:

```
struct ddctl {
        daddr_t    dc_bno;     /* Block number           */
        waddr_t    dc_buff;    /* Buffer to return data  */
        int        dc_off;     /* Offset                 */
        long       dc_count;   /* Count                  */
        long       dc_name;    /* Physical device name   */
        int        dc_size;    /* Size                   */
        int        dc_type;    /* Cache type             */
};

#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/ddcntl.h>

fdes = open(DISKDEV, O_RDWR);
ioctl(fdes, cmd, &ddcntl)
```

Following is a description of the available ioctl requests:

DC_ACACHE	Adds a cache to a logical device dc_name (for logical device caching). dc_count is the number of cache buffers, dc_size is the size of each buffer, and dc_type is the type of cache (DDBMR or DDSSD).
DC_AFLW	Adds the dc_bno block on the dc_name device to the flaw map.
DC_DFLW	Removes the dc_bno block on the dc_name device from the flaw map.
DC_DOWN	Configures the dc_name device to DOWN. This causes any I/O requests to that device to fail and return an error.

`DC_GO`	Restarts the `dc_name` device.
`DC_MAINT`	Places slice `dc_count` of the logical device specified in `dc_off` into or out of maintenance mode, depending on whether `dc_type` is nonzero or 0. You can use this command only on slices that reside on physical devices that are configured `DOWN` or `RONLY`. The call sets bit `SI_OFLD` in the `si_flags` word of the size structure associated with the slice (see the `iobuf.h` file). Bit `SI_OFLD` can be read by using the `DC_MAP` system call associated with the `disk` device. To clear the bit, either issue another `DC_MAINT` call or configure `UP` the physical device that contains the slice. The `fsoffload`(8) command uses the `DC_MAINT` call.
`DC_RCACHE`	Removes all cache from a logical device (for logical device caching).
`DC_READ`	Reads the number of bytes (which must be a sector multiple) specified in `dc_count` to the address specified in `dc_buff` from the `dc_name` device.
`DC_RFLW`	Replaces a flawed block with a new spares block.
`DC_RIOBUF`	Returns the `iobuf` structure, which is defined in the `sys/iobuf.h` include file, for the device name in `dc_name` into the user's buffer at `dc_buff`.
`DC_RMAP`	Returns the `ldmap` entry and slice entry for the logical device specified by the minor device number in `dc_off` to the buffer in `dc_buff`.
`DC_RONLY`	Sets device `dc_name` so that new space will not be allocated on that device.
`DC_RTAB`	Returns to the buffer in `dc_buff` the `iobuf` structures for all configured physical devices.
`DC_SSDOFF`	Configures the SSD channel specified in `dc_off` to `DOWN`.
`DC_SSDON`	Configures the SSD channel specified in `dc_off` to `UP`.
`DC_SSDTAB`	Returns the SSD control table (`ssdconf`) to the buffer in `dc_buff`.
`DC_SSTHRSH`	Sets the SSD threshold between synchronous and asynchronous transfers to `dc_count` sectors.
`DC_STOP`	Stops the `dc_name` device. This causes all I/O requests to that device to be queued until the device is configured up through `DC_UP`.
`DC_SYNCALL`	Flushes all logical device caches to disk.
`DC_UP`	Configures the `dc_name` device to `UP`.
`DC_WRITE`	Writes the number of bytes (which must be a sector multiple) specified in `dc_count` to the address specified in `dc_buff` from the device `dc_name`.

The possible errors for the system calls and probable causes are as follows:

[EEXIST]	This error appears when it is detected that a flaw already exists for block dc_bno (DC_AFLW).
[EFAULT]	This error appears when one of the following conditions has occurred: • The address of the ddcntl structure is out of range. • The logical device number dc_off is too big (DC_RMAP, DC_MAINT). • The slice number in dc_count is too big (DC_MAINT). • The buffer address dc_buff is out of range.
[EINVAL]	This error appears when one of the following conditions has occurred: • The block number to be flawed is out of range (DC_AFLW). • The slice to be placed into maintenance mode is not DOWN or RONLY (DC_MAINT). • The SSD channel number was not found (DC_SSDOFF, DC_SSDON). • Undefined command.
[ENOENT]	This error appears when one of the following conditions has occurred: • The name of the physical device dc_name is not found. • No spares device is configured (DC_AFLW, DC_DFLW, DC_RFLW). • The flaw is not found (DC_DFLW, DC_RFLW).
[ENOSPC]	This error appears when no spares are left (DC_AFLW, DC_RFLW).

NOTES

Only the super user can use the /dev/disk interface.

CRAY EL systems do not support SSD devices.

FILES

/dev/disk	Physical disk interface file
/usr/include/sys/ddcntl.h	Control structure definition for the ioctl system call
/usr/include/sys/fcntl.h	Structure definition for fcntl
/usr/include/sys/iobuf.h	Structure definition for iobuf
/usr/include/sys/types.h	Data type definition file
cf/conf.*SN*.c	System configuration file (*SN* is the mainframe serial number)

SEE ALSO

`dsk`(4)

`ioctl`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`pddconf`(8), `pddstat`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

dm – Kernel to data migration daemon communications interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /dev/dm/mig0 special file contains messages for dmdaemon(8), the data migration daemon. Messages are sent to the data migration daemon when the kernel recognizes the need for a file recall, the removal of an offline file, or the cancellation of a file recall.

The data migration daemon replies to the kernel requests through the same special file. These replies, currently used only for a recall request, indicate either a successful completion of the recall or an error.

The format of the requests and replies on this device are defined in sys/dmkreq.h, as follows:

```
struct dmkr {
        int     kr_magic;         /* for verification purposes   */
        int     kr_req;           /* request                     */
        int     kr_rep;           /* reply                       */
        int     kr_error;         /* error number                */
        int     kr_seq;           /* sequence number             */
        struct  dm_dvino kr_dvi;  /* device/inode of the file    */
        struct  offhdl kr_hdl;    /* file handle of the file     */
} mc_msg;
```

The following are the available ioctl requests, as defined in the sys/dmkreq.h include file:

MIG_DEBUG	Sets a debug mode that echoes messages to the system console.
MIG_NBIO	Enables or disables nonblocking I/O; currently unused.
MIG_NREAD	Returns the number of bytes available to be read.

NOTES

Only one process at a time is permitted to have this device open.

FILES

/dev/dm/mig0	Message file for kernel to data migration communication

SEE ALSO

dmdaemon(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`dsk` – Disk drive interface

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `/dev/dsk` file contains block special files that represent logical disk devices. A *logical disk device* is a collection of blocks on one or more physical disk or other logical disk devices.

The block special files in `/dev/dsk` are the interface to disk devices for the UNICOS file system. They have the major device number of the logical disk driver, `ldd`. See `ldd`(4) and `ldesc`(5).

The block special devices in `/dev/dsk` are made by using the `mknod`(8) command. Each device can reference one logical or physical disk device directly or more than one device indirectly.

The `mknod`(8) command is used as follows to create a logical disk inode:

`mknod` *name* b *major minor* 0 0 *path*

name	Name of the logical device.
`b`	The device is a block special device.
major	Major device number of the concatenated logical disk device driver. The driver is denoted by the name `dev_ldd` and is defined in `/usr/src/uts/c1/cf/devsw.c`.
minor	Minor device numbers must be unique among logical disk devices. Major device 0 is reserved for the `/dev/disk` control device (description follows).
`0 0`	Placeholders for future use.
path	Device path name. Path names for devices are full path names and are limited to 23 characters in length.

Two types of devices can be defined by using the `mknod` command: logical direct devices and logical indirect devices. A *logical direct* device indicates that the logical disk is comprised of exactly one disk slice. The path in the `mknod` parameter represents another block or character special device.

```
mknod /dev/dsk/usr b 34 20 0 0 /dev/pdd/usr
```

A *logical indirect* device indicates that the logical disk is comprised of more than one disk slice. The path in the `mknod` parameter represents a logical descriptor file. See `ldesc`(5).

```
mknod /dev/dsk/usr b 34 21 0 0 /dev/ldd/usr
```

The following `ioctl`(2) system calls are supported through the `/dev/disk` control device.

`ioctl`(*fd*, *cmd*, *arg*)

fd	Open file descriptor for `/dev/disk`.
cmd	*cmd* can be one of the following parameters:

`DC_ACACHE`	Adds cache to a logical device
`DC_RCACHE`	Removes cache from a logical device
`DC_RMAP`	Returns the `ldmap` structure for the logical device
`DC_SYNCALL`	Flushes the logical disk device cache to disk

arg A pointer to `struct ddctl`. The `ddctl` structure is defined in `sys/ddcntl.h`. The minor device number of the target device is specified in the `dc_off` field.

For a description of the physical characteristics of disk drives for systems with an IOS model E, see `diskspec`(7).

FILES

`/dev/dsk/*`

`/dev/ldd/*`

`/usr/include/sys/ldesc.h`

SEE ALSO

`disk`(4), `ldd`(4), `ldesc`(5), `mdd`(4), `pdd`(4), `ssd`(4), `ssdd`(4)

`diskspec`(7) (available only online) for IOS model E

`ddstat`(8), `mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`err` – Error-logging interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

Minor device 0 of the error-logging interface driver, `err`, is the interface between a process and the system's error-record collection routines. The `/dev/error` special file represents the `err` driver. A single process that has super-user permissions may open the driver for reading only. Each `read` operation causes an entire error record to be retrieved. If the `read` request is for less than the length of the record, the record is truncated.

FILES

`/dev/diag`

`/dev/error`

`/dev/MAKE.DEV`

`/usr/include/sys/erec.h`

SEE ALSO

`errfile`(5)

`dgdemon`(8), `errdemon`(8), `errpt`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

esd – External Semaphore Device Logical-layer Interface

SYNOPSIS

/dev/sfs

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The /dev/sfs device is used as the interface to the logical layer of the External Semaphore Device (ESD) driver. The logical layer of the ESD driver manages all semaphore lock assignment, heart-beat monitoring, and port table management.

The ioctl(2) system call is used to issue requests to the ESD driver.

The control structure used for the ioctl system call is defined in the sys/esd.h include file, as follows:

```
struct esdctl {
        word    esdf_narg[5];           /* Function arguments */
        word    esdf_reply;             /* Function reply area address */
};

/*
 * used for:    ESDF_ASGNSEMA
 *              ESDF_RELSEMA,
 *              ESDF_SETSEMA,
 *              ESDF_CLRSEMA,
 *              ESDF_TESTSEMA,
 *              ESDF_TSETSEMA,
 *              ESDF_SET_XCLR_SEMA,
 */
#define esdf_fs_id  esdf_narg[0]        /* Semaphore filesys/proc id word 0 */
#define esdf_fs_id1 esdf_narg[1]        /* Semaphore filesys/proc id word 1 */

/*
 * used for:    ESDF_RELSEMA,
 *              ESDF_SETSEMA,
 *              ESDF_CLRSEMA,
 *              ESDF_TSETSEMA,
 *              ESDF_TSETSEMA,
 *              ESDF_SET_XCLR_SEMA,
 */
#define esdf_semano esdf_narg[0]        /* Semaphore number */
#define esdf_fnflags esdf_narg[3]       /* Function flags */
                                        /* ESDF_ID = 0 => use esdf_semano    */
```

```
                                          /* ESDF_ID = 1 => use esdf_fs_id/id1 */

/*
 * used for:    ESDF_TSETSEMA,
 */
#define esdf_timeout esdf_narg[4]       /* Time out value (in clocks) */

/*
 * used for:    ESDF_REPORT
 *              ESDF_READHBEAT
 */
#define esdf_replysz esdf_narg[4]       /* Reply block size (in words) */
```

The first five words of the `esdctl` structure have different meanings, depending on the command being used. The `#define` statements following the definition of the `esdctl` structure provide an easy way of redefining the control words in lieu of using a `union`.

The general method of interfacing to the ESD driver involves passing in the handle for the requested semaphore by name or by semaphore number. When the semaphore name is supplied in `esdf_fs_id` and `esdf_fs_id1` fields of the `request` structure, the `ESDF_ID` flag must be set in the `esdf_fnflags` field. If semaphore number is supplied, the `ESDF_ID` flag must be 0. The `esdf_reply` field of the `esdctl` structure must be initialized to the address of a `reply` structure. The address of the `request` structure is passed in the `ioctl` call.

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

structure esdctl request;
structure esdrep reply;

if (fd = open("/dev/sfs", O_RDONLY)....
      .
      .
request.esdf_reply = (word)&reply;
ioctl(fd, cmd, &request)
```

On return, the `reply` structure contains information filled in based on the command type:

```
struct esdrep {
        word    esdr_semano;            /* semaphore number */
        word    esdr_result;            /* result code (-1 => error) */
        word    esdr_error;             /* error code */
        word    esdr_state;             /* previous/current state */
        word    esdr_last_port;         /* last port holding sema */
        word    esdr_portname;          /* last port (name) holding sema */
        struct  timeval esdr_time;      /* assignment time */
        word    esdr_avail;             /* # available user-assignable sema’s */
        word    esdr_used;              /* # used user-assignable sema’s */
};
```

A description of the available `ioctl` requests follows:

`ESDF_ASGNSEMA` Assigns a semaphore.

Requires: `esdf_fs_id/esdf_fs_id1`

Returns: `esdr_result`

0	= Success
-1	= Error (for code, see `esdr_error`)

`ESDF_RELSEMA` Releases a semaphore.

Requires: `esdf_fs_id/esdf_fs_id1`, or `esdf_semano`
`esdf_fnflags`

Returns: `esdr_result`

0	= Success
–1	= Error (for code, see `esdr_error`)

`ESDF_SETSEMA` Sets the semaphore to 1.

Requires: `esdf_fs_id/esdf_fs_id1`, or `esdf_semano`
`esdf_fnflags`

Returns: `esdr_result`

0	= Success
–1	= Error (for code, see `esdr_error`)

`ESDF_CLRSEMA` Sets the semaphore to 0.

Requires: `esdf_fs_id/esdf_fs_id1`, or `esdf_semano`
`esdf_fnflags`

Returns: `esdr_result`

0	= Success

		–1	= Error (for code, see `esdr_error`)
`ESDF_TESTSEMA`	Returns the current value of the semaphore.		
	Requires:	`esdf_fs_id`/`esdf_fs_id1`, or `esdf_semano` `esdf_fnflags`	
	Returns:	`esdr_result`	
		0	= Success
		–1	= Error (for code, see `esdr_error`)
`ESDF_TSETSEMA`	If the semaphore is currently 0, set it to 1; otherwise, return with a failure.		
	Requires:	`esdf_fs_id`/`esdf_fs_id1`, or `esdf_semano` `esdf_fnflags` `esdf_timeout`	
		0	= One attempt.
		>0	= Number of clocks periods to wait.
	Returns:	`esdr_result`	
		1	= Semaphore already set by this system
		0	= Success
		–1	= Error (for code, see `esdr_error`)
`ESDF_REPORT`	Reports All/Assigned Semaphores		
	Requires:	`esdf_replysz`	
`ESDF_READHBEAT`	Read Heart Beat Status		
	Requires:	`esdf_replysz`	
`ESDF_SET_XCLR_SEMA`	Sets semaphore unconditionally. Clear, if setting process exits.		
	Requires:	`esdf_fs_id`/`esdf_fs_id1`, or `esdf_semano` `esdf_fnflags`	
	Returns:	`esdr_result`	
		1	= Semaphore already set by this system
		0	= Success
		–1	= Error (for code, see `esdr_error`)

EXAMPLES

An example of assigning a semaphore directly follows:

```
/*
 * Initialize the 16-character cluster-unique name
 * to be assigned to this semaphore.  Care should be
 * taken to not use the same conventions as used to
 * identify filesystems.
 */
request.esdf_fs_id  = ’01234567’;
request.esdf_fs_id1 = ’89ABCDEF’;

request.esdf_reply = (word)&reply;

if (ioctl(fd, ESDF_ASGNSEMA, &request) < 0) {
      .... error, ioctl refused....
}
```

At this point, the `ioctl` has returned an `esdrep` structure that contains the fields at the `esdf_reply` address:

```
struct esdrep {
  word  esdr_semano;            /* semaphore number          */
  word  esdr_result;            /* result code (-1 => error) */
  word  esdr_error;             /* error code                */
  word  esdr_state;             /* previous/current state    */
  word  esdr_last_port;         /* last port holding sema    */
  word  esdr_portname;          /* last portname holding sema*/
  struct timeval esdr_time;     /* assignment time           */
  word  esdr_avail;             /* # avail user-assignable   */
                                /* sema’s                    */
  word  esdr_used;              /* # used user-assgn sema’s  */
};

Where:
  esdr_semano              The semaphore # assigned
   esdr_result              0 for success, -1 for some error condition
   esdr_error               If there was an error, the expanded error
                            codes are defined in sys/esd.h
   esdr_state               The ‘state’ field from the response word
                            returned from the hardware semaphore box
                            after ‘clear’ing the semaphore, not very
                            interesting at user-level
   esdr_last_port           The port #    of the assigning system
   esdr_portname            The port name of the assigning system
   esdr_time                The time the assignment occurred
   esdr_avail               The # of hardware semaphores available to
                            be assigned
   esdr_used                The # of hardware semaphores already assigned
```

An example of releasing a semaphore directly follows:

```
/*
 * Indicate which semaphore is to be released
 */
request.esdf_semano  = a_semaphore_#;
request.esdf_fnflags = 0;

request.esdf_reply   = (word)&reply;

if (ioct.l(fd, ESDF_RELSEMA, &request) < 0) {
      .... error, ioctl refused....
}
```

At this point, the `ioctl` has returned an `esdrep` structure in `reply` with the following fields:

esdr_semano	The semaphore # released
esdr_result	0 for success, -1 for some error condition
esdr_error	If there was an error, the expanded error codes are defined in sys/esd.h
esdr_state	0
esdr_last_port	The port # of the releasing system
esdr_portname	The port name of the releasing system
esdr_time	The time the assignment occurred
esdr_avail	The # of hardware semaphores available to be assigned
esdr_used	The # of hardware semaphores already assigned

An example of setting a semaphore directly follows:

```
/*
 * Indicate which semaphore is to be set
 */
request.esdf_semano  = a_semaphore_#;
request.esdf_fnflags = 0;

request.esdf_reply   = (word)&reply;

if (ioct.l(fd, ESDF_SETSEMA, &request) < 0) {
      .... error, ioctl refused....
}
```

At this point, the `ioctl` has returned an `esdrep` structure in `reply` with the following fields:

`esdr_semano`	The semaphore # just set
`esdr_result`	0 for success, -1 for some error condition
`esdr_error`	If there was an error, the expanded error codes are defined in sys/esd.h
`esdr_state`	The ‘state’ field from the response word returned from the hardware semaphore box after ‘set’ing the semaphore, indicates whether it was set or clear before the current set operation
`esdr_last_port`	The port # of the last system to change the semaphore
`esdr_portname`	The port name of the last system to change the semaphore
`esdr_time`	0
`esdr_avail`	0
`esdr_used`	0

An example of clearing a semaphore directly follows:

```
/*
 * Indicate which semaphore is to be cleared
 */
request.esdf_semano  = a_semaphore_#;
request.esdf_fnflags = 0;

request.esdf_reply   = (word)&reply;

if (ioct.l(fd, ESDF_CLRSEMA, &request) < 0) {
      .... error, ioctl refused....
}
```

At this point, the `ioctl` has returned an `esdrep` structure in `reply` with the following fields:

esdr_semano	The semaphore # just cleared
esdr_result	0 for success, -1 for some error condition
esdr_error	If there was an error, the expanded error codes are defined in sys/esd.h
esdr_state	The ‘state’ field from the response word returned from the hardware semaphore box after ‘clear’ing the semaphore, indicates whether it was set or clear before the current clear operation
esdr_last_port	The port # of the last system to change the semaphore
esdr_portname	The port name of the last system to change the semaphore
esdr_time	0
esdr_avail	0
esdr_used	0

NOTE: A clear request is honored only if the semaphore was set by the same system that issues the clear request.

An example of testing a semaphore directly follows:

```
/*
 * Indicate which semaphore is to be tested
 */
request.esdf_fs_id  = ’01234567’;
request.esdf_fs_id1 = ’89ABCDEF’;
request.esdf_fnflags = ESDF_ID;

request.esdf_reply = (word)&reply;

if (ioct.l(fd, ESDF_TESTSEMA, &request) < 0) {
      .... error, ioctl refused....
}
```

At this point, the `ioctl` has returned an `esdrep` structure in `reply` with the following fields:

`esdr_semano`	`The semaphore # just cleared`
`esdr_result`	`0 for success, -1 for some error condition`
`esdr_error`	`If there was an error, the expanded error codes are defined in sys/esd.h`
`esdr_state`	`The ‘state’ field from the response word returned from the hardware semaphore box after ‘test’ing the semaphore, indicates whether it was set or clear`
`esdr_last_port`	`Port # of the last system to change the semaphore`
`esdr_portname`	`Port name of the last system to change the semaphore`
`esdr_time`	`0`
`esdr_avail`	`0`
`esdr_used`	`0`

NOTES

Only the super user can use the `/dev/sfs` interface.

FILES

`/dev/sfs`	External Semaphore Device Logical-layer Interface

SEE ALSO

`ioctl`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`FDDI` – ANSI Fiber Distributed Data Interface

IMPLEMENTATION

Cray PVP systems with an IOS model E

DESCRIPTION

The FDDI interface (or FDDI driver) drives an ANSI standard Fiber Distributed Data Interface (FDDI). Application processes use the `FDDI` driver by means of the standard UNICOS system calls (that is, `close`(2), `ioctl`(2), `open`(2), `read`(2), `reada`(2), `write`(2), and `writea`(2)). Each of the special files in a `/dev/fddi`*n*/* directory represents all of the logical paths on one physical network interface. Each occurrence of a `/dev/fddi`*n*/* directory represents a physical network interface.

By convention, FDDI file names have the following format:

`/dev/fddi`*n*`/fd`*xx*

n	Physical interface number
xx	Logical path number

All FDDI I/O is raw; that is, the user process is locked in memory as data moves directly between the user buffer and the network. Therefore, user buffers must be word-aligned, and their length must be in 8-byte multiples.

Cray FDDI supports station management (SMT), version 6.2.

FDDI Fundamentals

FDDI is a 100-Mbit/s token-ring network that is used as a high-performance interconnection among computers and peripheral equipment as well as a high-speed backbone network for medium performance local area networks (LANs). FDDI uses fiber-optic technology as its transmission medium and can be configured to support a sustained transfer rate of approximately 80 Mbit/s (10 Mbyte/s). FDDI can interconnect many nodes on a ring distributed over distances of several kilometers in length. FDDI can support rings made up of 1000 physical connections that span a total fiber path length of 200 km. Each point-to-point link that makes up the ring can be a maximum of 2 km in length for fiber-media applications. Other media technologies, such as copper twisted-pair, are being studied but have no standards published as yet. Distance limits and other characteristics of the ring extent are different for these other media types.

An FDDI ring consists of a set of stations logically connected as a serial string of stations and media to form a closed loop. Information is transmitted sequentially from one station to the next; each station regenerating and repeating the information. The station serves as a way of attaching one or more devices to the network to communicate with other devices on the network.

FDDI, as it is defined by the American National Standards Institute, is divided into three layers: physical layer (PL), data link layer (DLL), and station management (SMT). Each layer defines part of FDDI and is kept as independent of the other layers as possible.

The PL is divided into two sublayers: physical medium dependent (PMD) and physical layer protocol (PHY). The PMD defines and characterizes the fiber-optic drivers and receivers, media-dependent encoding requirements, cables and connectors (cable plant), power budgets, optical-bypass provisions, and all other physical hardware-related characteristics. The PHY provides connection between the PMD and DLL. PHY's responsibilities include clock synchronization with incoming code-bit stream, encoding and decoding of code-bit stream into a symbol stream for use by the upper layers, and media conditioning and initializing.

The DLL is divided into two sublayers: media access control (MAC) and logical link control (LLC). The MAC provides fair and deterministic access to the media, address recognition, and generation and verification of frame check sequences. Its primary function is the delivery of frames from station to station, including frame transmission, repetition, and removal. The LLC provides a common peer-to-peer protocol that facilitates the transfer of information and control between any pair of DLL service access points on the FDDI ring.

Station management (SMT) provides the control necessary to manage the processes that occur in all of the FDDI layers such that each station may work cooperatively on the FDDI ring. SMT provides services such as station insertion and removal, station initialization, configuration control, fault isolation and recovery, station isolation, statistics collection, and address administration.

FDDI User Interface

As with any network interfaces on Cray Research systems with an IOS model E systems, when the first logical path is opened on a FDDI interface, the physical channel is configured up. For FDDI, this implies going through physical connection management (PCM) to connect to its adjacent stations, usually referred to as its upstream and downstream neighbors. When the last logical path on a FDDI interface is closed, the physical channel is configured down, which causes the Cray FDDI interface to disconnect from its neighbors.

Before configuring the channel interface, the microprocessor (High Performance microController, or HPC) on the channel adapter must first be downloaded with its microcode. If any I/O operation is tried on the device before the microcode has been downloaded, an error will occur. Automatic configuring of the channel on the first `OPEN` occurs only after the channel adapter has been downloaded.

From the user's perspective, the *frame* is the basic unit of information to and from the network. The maximum frame size on an FDDI network is 4500 bytes, which includes 2 bytes of preamble, 1 byte of start delimiter (SD), 1 byte of frame control (FC), 6 bytes of destination MAC address (DA), 6 bytes of source address (SA), 4478 bytes of Information (INFO), 4 bytes of frame check sequence (FCS), and 2 bytes of ending delimiter/frame status (ED/FS). The FC, DA, and SA fields compose the *MAC header*. The FC byte identifies the frame's type (for example, it identifies the frame as being a token, a MAC frame, a LLC frame, or a SMT frame). MAC addresses used on FDDI are IEEE 48-bit (Ethernet/Canonical) addresses; however, the FDDI MAC standard requires the Ethernet addresses to be in Non-Canonical form (or MSB form) when placed onto the media. FDDI (MSB) form implies that each byte within the 48-bit address is end-for-end bit swapped (0xCC to 0x33). To make this transparent to users, the Cray FDDI interface hardware performs this translation on each frame transmitted or received; therefore, users always see true Ethernet addresses.

On writes, the user must compile a frame made up of the MAC header and the INFO field, but not the FCS or the ED/FS. When the frame is placed on the physical media, the hardware will append the PA, SD, FCS, and ED/FS fields. Therefore, the amount of data that the user is allowed to write is a maximum of 4491 bytes, plus however many pad bytes have been programmed. (Pad bytes are described later in this subsection.) Usually, 3 bytes of pad are before the FC byte when IP is the protocol being run over FDDI. The buffer that holds the frame to be transmitted must be on a Cray word boundary, even though the number of bytes written does not have to be a multiple of Cray words.

On reads, users may post reads of any size, as long as the buffer is aligned on a Cray word boundary. For SMT frames received from the network, if the buffer is large enough, a user will receive the entire FDDI frame from the media, including the 4 bytes of FCS. Users must be aware that these 4 bytes of FCS are in the buffer, and if necessary, subtract 4 from the length to compensate for it. For LCC frames, the cyclic redundancy check (CRC) is stripped off.

Internet protocol (IP) datagrams and address resolution protocol (ARP) requests and replies are sent and received over the FDDI interface. Both of these protocols use the LLC service of FDDI. All LLC services over FDDI use 802.2 LLC (for related documents, see the SEE ALSO section). RFC 1390 defines the specific encapsulation of IP datagrams and ARP requests and replies within an FDDI frame by using 802.2 LLC SNAP. The use of this encapsulation yields a frame in which the beginning of the IP header is not aligned on a Cray word boundary. This is not a desirable situation for the Cray TCP/IP implementation; therefore, to assist the protocol stacks in frame processing, the Cray FDDI hardware can strip and append a set of pad bytes to each frame transmitted or received from the network. The number of pad bytes is programmable from 0 to 7. For IP, the desired number of pad bytes is 3. If padding is enabled, users must compile frames with the pad bytes on writes, and they will receive frames with the pad bytes on reads. The hardware strips off the pad bytes before frame transmission.

Protocols such as ARP must be able to determine the IEEE 48-bit address of the FDDI interface to place this address in ARP replies from other hosts. To determine the MAC address of the Cray FDDI interface, users may do an `ioctl FDC_GET` request (see `FDIO_GETSET`) or an `ioctl COMM_IOC_GETULA` request (see `NETULA`), both of which have versions that the UNICOS kernel can use.

For more information about the `FDDI` standard, see the ANSI documents listed in the SEE ALSO section.

Station Management Assistance

SMT is divided into two major categories: connection management (CMT) and Frame-based Management. CMT is further divided into five separate entities: configuration management (CFM), entity coordination management (ECM), ring management (RMT), link error monitor (LEM), and physical connection management (PCM). The ANSI standard defines how to implement these different parts of SMT; however, the manufacturer of FDDI hardware must determine the implementation of all these different forms of station management.

In the Cray implementation, SMT is a destributed process. The Cray Research mainframe handles all frame-based management in the form of a daemon called SMTD. Connection management (CMT) is handled on the channel adapter itself, using a microprocessor (called the HPC) that runs microcode that manipulates the FDDI chipset hardware and also maintains all of the different software state machines needed to perform CMT.

In both cases, the FDDI driver has hooks to support the efforts of the SMTD and the HPC (for example, the driver maintains a structure for each physical FDDI interface called `smtinfo`). In this structure, data is kept on behalf of the SMTD. The data consists of such things as upstream and downstream neighbor addresses, MAC availability information, results of the Duplicate Addresses Test that the RMT performs in the HPC, and a part of the SMT time-stamp value. To access these values, the SMT uses the `ioctl` interface.

The mainframe also must have some control over the microcode that is running in the HPC on the channel adapter. It needs this for two reasons. First, the SMTD must be able to retrieve information from both the state machines and the physical FDDI chipset on the channel adapter to respond to SMT frames received from other stations on the FDDI network. To do this, an interface that makes the HPC look like a set of readable and writable registers to the mainframe was designed. Using this interface (which also uses `ioctl` requests), the SMTD can read or write some of the key registers on the channel adapter at virtually any time.

Second, when certain events occur on the channel adapter or the FDDI ring itself, the HPC notifies the mainframe by sending an *unsolicited interrupt*. Several events can cause one of these interrupts; some events need immediate action by the driver, and others do not. Some examples of events that cause interrupts to the mainframe are as follows:

- Changes in MAC availability
- The ring recovering from a TRACE (which is an FDDI term for media reconfiguration)
- The overflow of a 32-bit SMT time-stamp value

When these events occur, the FDDI driver takes the necessary action(s). There are times when the mainframe must force the CMT state machines to a given state (for example, if the mainframe wants to disconnect from the FDDI network, it must tell the HPC's PCM code to disconnect). It does this by use of another interface, which is similar to the register read-write interface, called the *HPC signal interface*. Again, by using `ioctl` requests, the mainframe can send a signal to the HPC to perform some action. Examples of these signals are Duplicate Address Test Failed (sent by the SMTD), `EC_CONNECT` (which causes a port to connect to its neighbor), and `EC_DISCONNECT` (which causes a port to disconnect from its neighbor).

Parameter File

At boot time, the FDDI driver is configured by using a parameter file. The following example shows the syntax for parameters entered in the parameter file:

```
2 fdmaxdevs;                      /* max no. of FDDI interfaces */
16 fdmaxpaths;                    /* max no. of lpaths per interface */
fddev 0 {                         /* first interface */
        treq 10;                  /* FDDI TREQ in milliseconds */
        padcnt 3;                 /* no. of pad bytes */
        maxwrt 10;                /* max no. of write requests to IOS */
        maxrd 10;                 /* max no. of read requests to IOS */
        iopath {
                cluster 0;
                eiop 1;
                channel 034;
        }
        logical path 0 {
                rft SMT;         /* want to receive SMT frames */
                read timeout 20; /* read time-out in seconds */
        }
        logical path 1 {
                rft ALL;         /* want to receive ALL frame types */
                read timeout 10;
        }
        logical path 5 {
                rft LLC;         /* want to receive LLC frames */
                read timeout 60;
        }
}
fddev 1 {                        /* second interface */
        treq 30;
        padcnt 0;
        maxwrt 5;
        maxrd 5;
        iopath {
                cluster 0;
                eiop 1;
                channel 036;
        }
        logical path 0 {
                read timeout 30;
        }
        logical path 1 {
                read timeout 15;
        }
}
```

The only mandatory parameter for FDDI is `iopath`. Without this parameter, the driver does not know of the physical location of the FDDI interface. If you omit this parameter, an open on any logical path on that device will receive a `FDER_NOCHAN` error.

FDDI `ioctl` Requests

Several `ioctl` requests are available, and they have the following format:

```
#include <sys/netdev.h>
#include <sys/fd.h>
#include <sys/fdsys.h>

ioctl (fildes, command, arg)
int fildes;      /* file descriptor returned on open */
int command;    /* request code (FDC_XXX in sys/fd.h) */
char *arg;      /* fdioreq, commstreq, or netula */
```

When an `ioctl` request is performed, the *arg* argument of the system call must point to either a `fdioreq` structure or a `commstreq` structure. Each of these structures have pointers to buffers that hold the actual `ioctl` data. These buffers must be of sufficient size to hold the data requested. If the request will pass back multiple structures of a given type, the buffer must be large enough to hold all instances of the requested structure.

The following shows the format of the structures used in the `ioctl` request. You can find those structures that are not shown in the various header files listed previously.

```
struct macaddr {
        uint         :16,  /* unused */
                ieee :48;  /* IEEE (Canonical) form */
        uint         :16,  /* unused */
                fddi :48;  /* FDDI (MSB) form */
};

struct netula {
        long    addr;      /* 48-bit address (Canonical form) */
};

struct commstreq {
        int     sfunc;     /* status request subfunction */
        char    *sbuf;     /* status buffer pointer */
        int     dev;       /* device number (-1 means all devices) */
        int     lpath;     /* logical path (-1 means all paths) */
        uint    slen;      /* status buffer length (bytes) */
        uint    epoch;     /* incremented every configuration change */
};
```

```
struct fdioreq {
        char    *buf;       /* buffer pointer */
        int     len;        /* buffer length */
        int     param;      /* parameter */
};

struct fdio_echo {
        char    data [FD_MAXECHO];
};

struct fdio_loadmicro {
        char    binary [FD_MAXLOAD];

};

struc fdio_dumpsm {
       char    data [FD_MAXDUMPSM]

};
struct fdio_getset {
        int     ieee_mac;  /* fd->mac.ieee (RO) */
        int     fddi_mac;  /* fd->mac.fddi (RO) */
        int     errno;     /* lp->errno (RO) */
        int     err;       /* lp->err (RO) */
        int     des;       /* lp->des (RO) */
        int     fsw;       /* lp->fsw (RO) */
        int     TREQ;      /* fd->TREQ (RW) */
        int     cc;        /* fd->cc (RW) */
        int     padcnt;    /* fd->padcnt (RW) */
        int     maxwrt;    /* fd->maxwrt (RW) */
        int     maxrd;     /* fd->maxrd (RW) */
        int     opt;       /* lp->opt (RW) */
        int     rft;       /* lp->rft (RW) */
        int     rtmo;      /* lp->rtmo (RW) */
};

struct fdio_smt_timestamp {
        int     hi_32;     /* Upper 32 bits maintained by driver */
};

struct fdio_dad_results {
        int     results;     /* results of Duplicate Addr Test */

};
```

```
struct fdio_mac_neighbors {
        struct  macaddr una; /* upstream neighbor */
        struct  macaddr dna; /* downstream neighbor */
};

struct fdio_hpc_reg_data {
        int     size;     /* size of access (none, byte, word, long) */
                             /* F_HPC_SIZE_NONE, F_HPC_SIZE_BYTE */
                             /* F_HPC_SIZE_WORD, F_HPC_SIZE_LONG */
        int     page;        /* memory page address */
                             /* FD_HPC_PAGE_SMT, FD_HPC_PAGE_RMT */
                             /* FD_HPC_PAGE_PORT1, FD_HPC_PAGE_PORT2 */
                             /* FD_HPC_PAGE_BMAC, FD_HPC_PAGE_PLAYER1 */
                             /* FD_HPC_PAGE_PLAYER2, FD_HPC_PAGE_DIAG */
        int     addr;        /* HPC register address (0x00 - 0xff) */
        int     data;        /* read or write data (right justified) */
};

struct fdio_signal_hpc {
        int     signo;       /* signal number to HPC */
                             /* F_HPC_SIG_RMT_DAD_FAIL */
                             /* F_HPC_SIG_RMT_DAD_PASS */
                             /* F_HPC_SIG_RMT_EC_DISCONNECT */
                             /* F_HPC_SIG_RMT_EC_CONNECT */
                             /* F_HPC_SIG_RMT_EC_PTPASS */
                             /* F_HPC_SIG_RMT_SYS_RESET */
};

struct fdio_hpc_info {
        uchar   smt_00[2];    /* Working Register I */
        uchar   smt_02[2];    /* Working Register J */
        uchar   smt_04[2];    /* Working Register P */
        uchar   smt_06;       /* Microcode Revision */
        uchar   smt_07;       /* Hardware Configuration */
        uchar   smt_08;       /* SMT State */
        uchar   smt_09;       /* SMT Configuration */
        uchar   smt_0A;       /* Interrupt Summary */
        uchar   smt_0B;       /* Interrupt Mask */
        uchar   smt_0C[2];    /* SMT Events */
        uchar   smt_0E[2];    /* SMT Event Mask */
        uchar   smt_10[4];    /* SMT Timestamp */
        uchar   smt_14[2];    /* SYSTIM Counter */
        uchar   smt_16;       /* Timer Events */
        uchar   smt_17;       /* Timer Event Mask */
uchar   rmt_00;       /* RMT State */
uchar   rmt_01;       /* RMT Status */
uchar   rmt_02[2];    /* RMT Timer */
uchar   rmt_04[2];    /* RMT Events */
uchar   rmt_06[2];    /* RMT Event Mask */
uchar   rmt_08[2];    /* Reserved word */
uchar   rmt_0A[2];    /* RTT - Restricted Token Timeout */
uchar   rmt_0C[2];    /* T_request value */
uchar   mac_0E[2];    /* Late_Ct (Late Count) */
uchar   mac_10[4];    /* Token_Ct (Token Count) */
uchar   mac_14[4];    /* TX_Ct (Transmit Count) */
uchar   mac_18[4];    /* NotCopied_Ct (Not Copied Count) */
uchar   mac_1C[4];    /* Frame_Ct (Frame Copied Count) */
uchar   mac_20[4];    /* Lost_Ct (Lost Frame Count) */
uchar   mac_24[4];    /* Error_Ct (Error Isolated Count) */
uchar   mac_28[4];    /* RX_Ct (Receive Count) */
uchar   mac_2C[4];    /* Ring_Ct (Ring Recovery Count) */
uchar   mac_30[4];    /* TVX_Ct (TVX Expiration Count) */
uchar   mac_34[4];    /* Latency_Ct (Ring Latency Count) */
uchar   port1_00;     /* Port 1 - PCM state */
uchar   port1_01;     /* Port 1 - PCM-PSC Index */
uchar   port1_02;     /* Port 1 - BREAK Count */
uchar   port1_03;     /* Port 1 - PCM Index at last BREAK */
uchar   port1_04[2];  /* Port 1 - Transition Table Pointer */
uchar   port1_06[2];  /* Port 1 - Current Line State */
uchar   port1_08[2];  /* Port 1 - Reserved Word */
uchar   port1_0A[2];  /* Port 1 - Start Value of TPC Counter */
uchar   port1_0C[2];  /* Port 1 - TCP Counter */
uchar   port1_0E[2];  /* Port 1 - Address of TPC Service Routine */
uchar   port1_10[2];  /* Port 1 - PSC Connection Policy */
uchar   port1_12[2];  /* Port 1 - PSC Rval */
uchar   port1_14[2];  /* Port 1 - PSC Tval */
uchar   port1_16;     /* Port 1 - PSC Status (Neighbor porttype) */
uchar   port1_17;     /* Port 1 - LCT Fail */
uchar   port1_18[2];  /* Port 1 - CEM Policy */
uchar   port1_1A;     /* Port 1 - CEM State */
uchar   port1_1B;     /* Port 1 - PLAYER Configuration Reg value */
uchar   port1_1C;     /* Port 1 - LEM Reject Count */
uchar   port1_1D;     /* Port 1 - LER Estimate */
uchar   port1_1E;     /* Port 1 - LER Alarm */
uchar   port1_1F;     /* Port 1 - LER Cutoff */
uchar   port1_20[4];  /* Port 1 - LEM Count */
uchar   port1_24[4];  /* Port 1 - Elasticity Buffer Error Count */
uchar   port1_28[4];  /* Port 1 - LER Average */
uchar   port1_2C[4];  /* Port 1 - LER Delta */
uchar   port1_30;     /* Port 1 - ECM State */
uchar   port1_31;     /* Port 1 - ECM Path */
uchar   port1_32[2];  /* Port 1 - Reserved Word */
uchar   port1_34[2];  /* Port 1 - PORT Events */
uchar   port1_36[2];  /* Port 1 - PORT Event Mask */
uchar   port2_00;     /* Port 2 - PCM state */
uchar   port2_01;     /* Port 2 - PCM-PSC Index */
uchar   port2_02;     /* Port 2 - BREAK Count */
uchar   port2_03;     /* Port 2 - PCM Index at last BREAK */
uchar   port2_04[2];  /* Port 2 - Transition Table Pointer */
uchar   port2_06[2];  /* Port 2 - Current Line State */
uchar   port2_08[2];  /* Port 2 - Reserved Word */
uchar   port2_0A[2];  /* Port 2 - Start Value of TPC Counter */
uchar   port2_0C[2];  /* Port 2 - TCP Counter */
uchar   port2_0E[2];  /* Port 2 - Address of TPC Service Routine */
uchar   port2_10[2];  /* Port 2 - PSC Connection Policy */
uchar   port2_12[2];  /* Port 2 - PSC Rval */
uchar   port2_14[2];  /* Port 2 - PSC Tval */
uchar   port2_16;     /* Port 2 - PSC Status (Neighbor porttype) */
uchar   port2_17;     /* Port 2 - LCT Fail */
uchar   port2_18[2];  /* Port 2 - CEM Policy */
uchar   port2_1A;     /* Port 2 - CEM State */
uchar   port2_1B;     /* Port 2 - PLAYER Configuration Reg value */
uchar   port2_1C;     /* Port 2 - LEM Reject Count */
uchar   port2_1D;     /* Port 2 - LER Estimate */
uchar   port2_1E;     /* Port 2 - LER Alarm */
uchar   port2_1F;     /* Port 2 - LER Cutoff */
uchar   port2_20[4];  /* Port 2 - LEM Count */
uchar   port2_24[4];  /* Port 2 - Elasticity Buffer Error Count */
uchar   port2_28[4];  /* Port 2 - LER Average */
uchar   port2_2C[4];  /* Port 2 - LER Delta */
uchar   port2_30;     /* Port 2 - ECM State */
uchar   port2_31;     /* Port 2 - ECM Path */
uchar   port2_32[2];  /* Port 2 - Reserved Word */
uchar   port2_34[2];  /* Port 2 - PORT Events */
uchar   port2_36[2];  /* Port 2 - PORT Event Mask */
uchar   bmac_87;      /* THSH1 - async priority 1 */
uchar   bmac_8B;      /* THSH2 - async priority 2 */
uchar   bmac_8F;      /* THSH2 - async priority 3 */
uchar   bmac_93;      /* T_max */
uchar   bmac_97;      /* TVX_value */
uchar   bmac_98[4];   /* T_negotiated */
uchar   cfm_state;    /* station CFM state */
```

;

The following table shows each of the requests and the structure formats used:

Request	`ioctl arg` parameter	`fdioreq.buf` or `commstreq.sbuf`
COMM_IOC_CDSTATS	commstreq	N/A
COMM_IOC_CLSTATS	commstreq	N/A
COMM_IOC_DSTATS	commstreq	commstat
COMM_IOC_LSTATS	commstreq	commstat
COMM_IOC_STATS	commstreq	commstat
COMM_IOC__GETULA	netula	N/A
COMM_IOC__KGETULA	netula	N/A
FDC_GET	fdioreq	fdio_getset
FDC_SET	fdioreq	fdio_getset
FDC_KGET	fdioreq	fdio_getset
FDC_KSET	fdioreq	fdio_getset
FDC_CDSTATS	commstreq	N/A
FDC_CLSTATS	commstreq	N/A
FDC_DSTATS	commstreq	commstat
FDC_LSTATS	commstreq	commstat
FDC_STATS	commstreq	commstat
FDC_ECHO	fdioreq	fdio_echo
FDC_ECHOSINK	fdioreq	fdio_echo
FDC_CLRDLF	fdioreq	N/A
FDC_SETDLF	fdioreq	N/A
FDC_LOADMICRO	fdioreq	fdio_loadmicro
FDC_DUMPSM	fdioreq	fdio_dumpsm
FDC_DSTRUCT	commstreq	fd_dev
FDC_LSTRUCT	commstreq	fd_lp
FDC_GETVARS	commstreq	fd_vars
FDC_SET_LLC_AV	none	N/A
FDC_CLR_LLC_AV	none	N/A
FDC_XCHG_DAD	fdioreq	fdio_dad_results
FDC_SET_MACNBRS	fdioreq	fdio_mac_neighbors
FDC_GET_MACNBRS	fdioreq	fdio_mac_neighbors
FDC_GET_DAD	fdioreq	fdio_dad_results
FDC_GET_HPC	fdioreq	fdio_hpc_info
FDC_OR_HPC_REG	fdioreq	fdio_hpc_reg_data
FDC_AND_HPC_REG	fdioreq	fdio_hpc_reg_data
FDC_RD_HPC_REG	fdioreq	fdio_hpc_reg_data
FDC_WR_HPC_REG	fdioreq	fdio_hpc_reg_data

Request	ioctl arg parameter	fdioreq.buf or commstreq.sbuf
FDC_SIGNAL_HPC	fdioreq	fdio_signal_hpc
FDC_RD_SMTTIME	fdioreq	fdio_smt_timestamp

The valid ioctl requests are as follows:

COMM_IOC_CDSTATS
: Clears the statistics associated with the device(s) specified by commstreq.dev. If commstreq.dev is a −1, all configured devices will be cleared.

COMM_IOC_CLSTATS
: Clears the statistics associated with the logical path(s) specified by commstreq.dev and commstreq.lpath. If commstreq.dev is a −1 and commstreq.lpath is a −1, all configured paths on all configured devices will be cleared.

COMM_IOC_DSTATS
: Returns the statistics for the device(s) specified by the commstreq.dev parameter. The format of the statistics returned is that of a commstat structure. If commstreq.dev is a −1, statistics for all configured devices will be returned.

COMM_IOC_LSTATS
: Returns the statistics for the logical path(s) specified by the commstreq.lpath parameter. The format of the statistics returned is that of a commstat structure. If commstreq.dev is a −1 and commstreq.lpath is a −1, statistics for all configured logical paths on all configured devices will be returned.

COMM_IOC_STATS
: Returns both device and logical path statistics associated with the device(s) and logical path(s) specified by commstreq.dev and commstreq.lpath. This is the same as FDC_DSTATS and FDC_LSTATS combined.

COMM_IOC_GETULA
: Gets the value of the IEEE Universal LAN address.

COMM_IOC_KGETULA
: Same as FDC_GETULA except that the kernel uses it.

FDC_GET
: Gets the current driver parameter settings and stores them in the fdio_getset structure to which fdioreq.buf points.

`FDC_SET`
: Sets driver parameters from the `fdio_getset` structure to which `fdioreq.buf` points. All driver parameters that are changed by this `ioctl` request reset to their boot-time values when the logical path is closed. Parameters that apply to the device as a whole, such as `TREQ`, are reset to their boot-time values when the last logical path is closed on the device.

`FDC_KGET`
: Same as `FDC_GET` except that the kernel uses it.

`FDC_KSET`
: Same as `FDC_SET` except that the kernel uses it.

`FDC_CDSTATS`
: Same as `COMM_IOC_CDSTATS`.

`FDC_CLSTATS`
: Same as `COMM_IOC_CLSTATS`.

`FDC_DSTATS`
: Same as `COMM_IOC_DSTATS`.

`FDC_LSTATS`
: Same as `COMM_IOC_LSTATS`.

`FDC_STATS`
: Same as `COMM_IOC_STATS`.

`FDC_ECHO`
: Sends the data to which `fdioreq.buf` points to the IOS I/O buffer. This is a simulated write operation that does not activate the channel. To read data back, post a read command on the device.

`FDC_ECHOSINK`
: Sends the data to which `fdioreq.buf` points to the IOS I/O buffer. This is a simulated write operation that does not activate the channel. Data cannot be read back by posting a read command on the device.

`FDC_CLRDLF`
: Clears the internal microcode download flag in the driver. When the flag is clear, the driver does not allow any I/O to be performed on the device. Also, when the flag is clear, the channel is not automatically configured on the first open.

`FDC_SETDLF`
: Sets the internal microcode download flag in the driver. When the flag is set, the driver allows I/O to be performed on the device. Also, when the flag is set, the channel is automatically configured on the first open.

`FDC_LOADMICRO`
: Sends the binary data to which `fdioreq.buf` points in the IOS I/O buffer. The IOP takes this binary data and loads it into the FCA-1's shared memory at the address specified in the binary data itself. After the reset signal is inactivated to the FCA-1, the HPC processor will begin to execute the code that was just downloaded.

`FDC_DUMPSM`
: Returns the contents of FCA-1 shared memory. The `fdioreq.param` specifies the address of the shared memory to be returned.

`FDC_DSTRUCT`
: Returns the contents of the `fd_dev` structures for the device(s) specified by `commstreq.dev`. This structure is the internal driver structure used for controlling each FDDI interface.

`FDC_LSTRUCT`
: Returns the contents of the `fd_lp` structures for the logical path(s) specified by `commstreq.dev` and `commstreq.lpath`. This structure is the internal driver structure used for controlling each logical path on each of the FDDI interfaces.

`FDC_GETVARS`
: Returns the contents of the `fd_vars` structure, which contains the maximum number of FDDI devices and logical paths.

`FDC_SET_LLC_AV`
: Sets the logical link control (LLC) available flag, `fd_dev.smt.llc_available`, for this device.

`FDC_CLR_LLC_AV`
: Clears the LLC available flag, `fd_dev.smt.llc_available`, for this device.

`FDC_XCHG_DAD`
: Exchanges the new results of the Duplicate Address Test with the current results that are stored in the Driver Device table. If the results are different, the driver will notify the HPC on the channel adapter of the new results.

`FDC_SET_MACNBRS`
: Sets the upstream and downstream MAC neighbor addresses from the `fdio_mac_neighbors` structure to which `fdioreq.buf` points.

`FDC_GET_MACNBRS`
: Returns the upstream and downstream MAC neighbor addresses. The format of the data returned is in the form of the `fdio_mac_neighbors` structure.

`FDC_GET_DAD`
: Returns the results of the Duplicate Address Test. The format of the data returned is in the form of the `fdio_dad_results` structure.

`FDC_GET_HPC`
: Returns the register information from the HPC on the channel adapter. The format of the data returned is in the form of the `fdio_hpc_info` structure.

`FDC_OR_HPC_REG`
: Logically ORs the contents of a particular HPC register on the channel adapter with the specified value. The format of the data to OR is in the form of the `fdio_hpc_reg_data` structure.

`FDC_AND_HPC_REG`
: Logically ANDs the contents of a particular HPC register on the channel adapter with the specified value. The format of the data to AND is in the form of the `fdio_hpc_reg_data` structure.

`FDC_RD_HPC_REG`
: Returns the contents of a particular HPC register on the channel adapter. The format of the data returned is in the form of the `fdio_hpc_reg_data` structure.

`FDC_WR_HPC_REG`
: Sets the contents of a particular HPC register on the channel adapter to the specified value. The format of the data to write is in the form of the `fdio_hpc_reg_data` structure.

`FDC_SIGNAL_HPC`
: Sends the specified signal number to the HPC on the channel adapter. The format of the data is in the form of the `fdio_signal_hpc` structure.

`FDC_RD_SMTTIME`
: Returns the high-order 32 bits of the 64-bit SMT time stamp that the driver maintains by the low-order 32 bits. The HPC on the channel adapter maintains the low-order bits, and they are available in the `fdio_hpc_info` structure. The format of the data is in the form of the `fdio_smt_timestamp` structure.

The `fdio_getset` structure contains the following fields:

`int ieee_mac`
: MAC address; IEEE (Canonical/Ethernet) form. (Ignored on `FDC_SET` and `FDC_KSET`.)

`int fddi_mac`
: MAC address; FDDI form. (Ignored on `FDC_SET` and `FDC_KSET`.)

`int errno`
: Error code returned to user (`errno`). (Ignored on `FDC_SET` and `FDC_KSET`.)

`int err`
: FDDI error code. (Ignored on `FDC_SET` and `FDC_KSET`.)

`int des`
: Detailed error status code from IOS. (Ignored on `FDC_SET` and `FDC_KSET`.)

`int fsw`
: Frame status word from last frame read on this logical path. (Ignored on `FDC_SET` and `FDC_KSET`.)

`int TREQ`
: `TREQ` value for this interface. `TREQ` is the value that is sent on all Claim frames from the `FDDI BMAC`. This is the requested value for the token rotation timer. (Default is 167 ms.)

`int cc`
: Copy criteria mask for this interface. For the specific values for this mask, see `sys/epackf.h`. (Default is `LLC` and `SMT`.)

`int padcnt`
: Pad count (default is 3 bytes) for this interface. This is the number of bytes that are removed from the start of each frame transmitted and inserted at the start of each frame received. This padding is required for protocols such as TCP/IP, which needs the IP header word aligned within the FDDI frame.

`int maxwrt`
: Maximum number of write requests (default is 10) allowed to IOS. After this many write requests are pending in the IOS, the driver queues up any further write requests.

`int maxrd`
: Maximum number of read requests (default is 10) that can be sent to the IOS. After this number of read requests is pending in the IOS, the driver queues any additional read requests.

`int opt`
: Options (default is `NONE`) for this logical path. Currently, two options (`NFRCHK` and `NERRLOG`) are defined for logical paths. The `NFRCHK` option causes the driver to bypass frame validity checking on write operations. This option can be useful for diagnostics. The `NERRLOG` option causes the driver to not log errors that occur on this logical path to the system error log. This option is useful for diagnostics that are causing errors intentionally and want to avoid having a record of those errors. For the bit definitions of each of these options, see the `FDLO_XXXX`, defined in `sys/fd.h`.

`int rft`
: Receive frame type mask (default is `NONE`) for this logical path. This field defines the types of frames to be received by this logical path. Frame types can be SMT frames, LLC frames, or any combination of frame types. However, any particular frame type can be registered to be received only by a single-logical path. If a logical path tries to register to receive a frame type that is already being received by another path, an error will occur. For the specific values for this mask, see the `EFOP_RFT_XXXX`, defined in `sys/epackf.h`.

`int rtmo`
: Read time-out value (in seconds) for this logical path; default is 60 seconds.

EXIT STATUS

The FDDI driver returns one of the following error codes in `errno` on an error. To obtain a more specific error code information, use a `FDC_GET ioctl` request and examine the `err` and `des` fields. For the mapping of the specific error codes to user error codes, see the table that follows.

The error codes and their meanings are as follows:

`EBUSY` The specified logical path is in use.

`EFAULT` An argument to a `ioctl` request is not valid.

`EINVAL` The driver has detected a parameter error.

`EIO` This error can be caused when a fatal I/O error occurs in the IOP, the FDDI MAC or LLC services are not available on a write, a frame type that was not valid was received from the network, the error bit is set in the frame status in a received frame on a read, or the actual transfer length does not equal the request transfer length on a write or read.

`ELATE` A request timed out.

`EPERM` The driver has detected an operation that requires super-user privileges by a user that does not have those privileges.

`ENXIO` The specified device or logical path does not exist or is not in an operational state.

The mainframe driver returns the following specific error codes:

`FDER_BADDR`
A `b_waddr` value that is not valid in the *buf* (bp) structure was detected on a read or write.

`FDER_BADHPCADDR`
A `FDC_OR_HPC_REG`, `FDC_AND_HPC_REG`, `FDC_RD_HPC_REG`, or `FDC_WR_HPC_REG` was issued with an HPC address that is not valid. The address must be less than or equal to 0x00ff.

`FDER_BADHPCPAGE`
A `FDC_OR_HPC_REG`, `FDC_AND_HPC_REG`, `FDC_RD_HPC_REG`, or `FDC_WR_HPC_REG` was issued with a HPC page address that is not valid. The page must be less than or equal to 0x07.

`FDER_BADHPCSIZE`
A `FDC_OR_HPC_REG`, `FDC_AND_HPC_REG`, `FDC_RD_HPC_REG`, or `FDC_WR_HPC_REG` was issued with a HPC size that is not valid. The size must be byte, word, or long.

`FDER_BADIOFLAG`
The `u_io` flag is not set for the user or the kernel making an `ioctl` request.

`FDER_BCOUNT`
A `b_count` value that is not valid in the *buf* (bp) structure was detected on a read or write.

`FDER_BUSY`
An open operation is tried on a logical path that is already open.

`FDER_CLSNOPEN`
A close operation is tried on a logical path that is not open.

`FDER_CONFINGDN`
An open operation is tried on a device that is in the process of being configured down.

`FDER_CONFUPER`
An open operation is tried on a device that did not configure up successfully.

FDER_COPYIN
: An error occurred when copying data from the user to the kernel.

FDER_COPYOUT
: An error occurred when copying data from the kernel to the user.

FDER_ESET
: The error bit is set in the received frame.

FDER_HALTED
: The I/O was halted.

FDER_IOCPARAM
: A parameter that was not valid was detected on an `ioctl` request.

FDER_IOCREQUEST
: A `ioctl` request that was not valid was made.

FDER_LLC
: An unsupported 802.2 LLC is detected.

FDER_LLCNAVAIL
: The LLC services are not available.

FDER_MACNAVAIL
: The MAC services are not available.

FDER_NOCHAN
: An open operation is tried on a device that has no physical channel defined in the configuration.

FDER_NOPEN
: A operation is tried on a logical path that is not open.

FDER_NORFT
: A read is posted on this path, but the path has not yet registered to receive any frame types.

FDER_NOTZUP
: A nonsuper user tried an `FDC_SET`.

FDER_NULLDA
: A null destination address (DA) in the FDDI frame was detected on a write.

FDER_OK
: No error (binary 0).

FDER_PACKLEN
: The length of an F-packet received from the IOS was incorrect.

FDER_PACKOUT
: An error was detected when sending an F-packet to the IOS.

`FDER_RANGE`
: A device or logical path index is out of range. Typically, error this occurs when the minor device number is created incorrectly.

`FDER_RCVFC`
: A frame type that was not valid was received from the network.

`FDER_RFTINUSE`
: The frame type that you want to register for a path is already registered to another logical path.

`FDER_WRITEFC`
: A frame control (FC) byte that is not valid in the FDDI frame was detected on a write.

`FDER_XNFRLEN`
: The actual transfer length does not equal the requested transfer length.

`FDER_UNSOLICITED`
: An unsolicited error has been received from the EIOP. This error may be asynchronous with the operation that the mainframe is currently performing on the channel.

`FDER_NOTLOADED`
: An attempt was made to open a logical path or to perform a read or write on a logical path before the channel adapter has been downloaded with microcode.

`FDER_ALRDYLOADED`
: An attempt was made to download the channel adapter with the download flag already set, meaning the adapter had been already loaded.

The IOS driver returns the following specific error codes:

`F_RSP_ACT_CRE8`
: Cannot create needed activity.

`F_RSP_BADCHN`
: Channel number is not valid.

`F_RSP_BADFCA1`
: `FCA1` mode is not valid.

`F_RSP_BADLEN`
: Requested transfer length is not valid.

`F_RSP_BADPATH`
: Logical path is not valid.

`F_RSP_BADREQ`
: Request code is not valid.

`F_RSP_BADTMO`
: Supplied time-out value is not valid.

`F_RSP_BAD_TYPE`
Bad packet type.

`F_RSP_CBREL_BAD`
CB_Release error.

`F_RSP_CBRES_BAD`
CB_Reserve error.

`F_RSP_CBNAVL`
Channel buffer not available.

`F_RSP_CH_DOWN`
Channel pair not configured up.

`F_RSP_CH_INITING`
Initialization of channel pair already in progress.

`F_RSP_CH_TERMING`
Termination of channel pair already in progress.

`F_RSP_CH_UP`
Channel pair already configured up.

`F_RSP_CLOSED`
Request aborted because of `CLOSE PATH` request.

`F_RSP_DVRTERM`
Driver terminated.

`F_RSP_FCA1_BAD`
`FCA1` hardware information is not valid.

`F_RSP_HALTED`
Request aborted because of `HALT I/O` request.

`F_RSP_HALT_FAIL`
Cannot `HALT I/O` in a driver activity.

`F_RSP_IOBMEM`
IOB memory not available.

`F_RSP_LOCMEM`
Local memory not available.

`F_RSP_LMIO_ADR`
Bad channel buffer address parameter in `LMIO` request.

`F_RSP_LMIO_DIR`
Bad transfer direction parameter in `LMIO` request.

`F_RSP_LMIO_HDWR`
Hardware error detected on `LMIO` attempt.

`F_RSP_LMIO_LEN`
: Bad word length parameter in `LMIO` request.

`F_RSP_LMIO_ORD`
: Bad ordinal parameter in `LMIO` request.

`F_RSP_OK`
: No error was detected.

`F_RSP_OVERRUN`
: Transferred more data than expected.

`F_RSP_ABORT_O`
: Output I/O buffer to FIFO abort.

`F_RSP_BUFMEM_O`
: Output I/O buffer single-or double-bit error.

`F_RSP_PAR_FETCH`
: Parity error during HPC fetch memory reference.

`F_RSP_IFCA1TMO`
: Input FCA-1 channel time-out.

`F_RSP_HPC_DOWNLOAD`
: HPC download failed.

`F_RSP_HPC_BADRESP`
: HPC sent bad response to mailbox command.

`F_RSP_HPC_TMO`
: Time-out occurred on a HPC mailbox request.

`F_RSP_PARITY_O`
: Output FIFO data/control/buffer parity error detected.

`F_RSP_PARITY_I`
: Input FIFO data/control/buffer parity error detected.

`F_RSP_PATHCLO`
: No open connection for this logical path.

`F_RSP_PATHOPN`
: Connection already open for this logical path.

`F_RSP_PKTLEN`
: Request packet length is not valid.

`F_RSP_RD_PKT_TMO`
: Read request packet timed out.

`F_RSP_RELMEM`
: `RELMEM` request failed.

F_RSP_TERM_FAIL
: Cannot terminate all driver activities.

F_RSP_TIMER_PAR
: Bad parameter on TIMER call.

F_RSP_TIMER_QUED
: Start a timer already on RTC queue.

F_RSP_TIMER_UNKNOWN
: An unknown timer failure.

F_RSP_TMIO_ADR
: Bad channel buffer address parameter in TMIO request.

F_RSP_TMIO_DIR
: Bad transfer direction parameter in TMIO request.

F_RSP_TMIO_HDWR
: Hardware error detected on TMIO attempt.

F_RSP_TMIO_LEN
: Bad word length parameter in TMIO request.

F_RSP_TMIO_NAVL
: Target memory channel not available.

F_RSP_TMIO_ORD
: Bad ordinal parameter in TMIO request.

The following shows the mapping of specific error codes and user error codes:

Specific Error Code	**User Error Code (errno)**
FDER_BADDR	EINVAL
FDER_BADHPCADDR	EINVAL
FDER_BADIOFLAG	EINVAL
FDER_BCOUNT	EINVAL
FDER_BUSY	EBUSY
FDER_CLSNOPEN	ENXIO
FDER_CONFINGDN	ENXIO
FDER_CONFUPER	ENXIO
FDER_COPYIN	EFAULT
FDER_COPYOUT	EFAULT
FDER_ESET	EIO
FDER_HALTED	EIO

FDER_IOCPARAM	EINVAL
FDER_LLCNAVAIL	ENXIO
FDER_MACNAVAIL	ENXIO
FDER_NOCHAN	ENXIO
FDER_NOPEN	ENXIO
FDER_NORFT	EIO
FDER_NOTZUP	EPERM
FDER_NULLDA	EINVAL
FDER_PACKLEN	EIO
FDER_PACKOUT	EIO
FDER_RCVFC	EIO
FDER_RFTINUSE	EIO
FDER_WRITEFC	EINVAL
FDER_XNFRLEN	EIO
F_RSP_ACT_CRE8	EIO
F_RSP_BADCHN	EINVAL
F_RSP_BADFCA1	EINVAL
F_RSP_BADLEN	EINVAL
F_RSP_BADPATH	EINVAL
F_RSP_BADREQ	EINVAL
F_RSP_BADTMO	EINVAL
F_RSP_BAD_TYPE	EINVAL
F_RSP_CBREL_BAD	EIO
F_RSP_CBRES_BAD	EIO
F_RSP_CBNAVL_BAD	EIO
F_RSP_CH_DOWN	EINVAL
F_RSP_CH_INITING	EINVAL
F_RSP_CH_TERMING	EINVAL
F_RSP_CH_UP	EINVAL
F_RSP_CLOSED	EIO
F_RSP_DVRTERM	EIO

F_RSP_FCA1_INFO_BAD	EIO
F_RSP_HALTED	EIO
F_RSP_HALT_FAIL	EIO
F_RSP_IOBMEM	EIO
F_RSP_LMIO_ADR	EIO
F_RSP_LMIO_DIR	EIO
F_RSP_LMIO_HDWR	EIO
F_RSP_LMIO_LEN	EIO
F_RSP_LMIO_ORD	EIO
F_RSP_LOCMEM	EIO
F_RSP_OVERRUN	EIO
F_RSP_PARITY	EIO
F_RSP_PATHCLO	EINVAL
F_RSP_PATHOPN	EINVAL
F_RSP_PKTLEN	EINVAL
F_RSP_RD_PKT_TMO	ELATE
F_RSP_RELMEM	EIO
F_RSP_SECDED	EIO
F_RSP_TERM_FAIL	EIO
F_RSP_TIMER_PAR	EIO
F_RSP_TIMER_QUED	EIO
F_RSP_TIMER_UNKNOWN	EIO
F_RSP_TMIO_ADR	EIO
F_RSP_TMIO_DIR	EIO
F_RSP_TMIO_HDWR	EIO
F_RSP_TMIO_LEN	EIO
F_RSP_TMIO_NAVL	EIO
F_RSP_TMIO_ORD	EIO

EXAMPLES

The following is an example of the usage of the `ioctl` request to alter driver parameters:

```
#include <sys/epackf.h>
#include <sys/fd.h>

struct fdioreq req;                  /* FDDI driver parameter structure */
int ret;                             /* Status returned from ioctl() req */
int fildes;                          /* FDDI device file descriptor */
struct fdio_getset getset;           /* getset structure */

req.buf = &getset;
req.len = sizeof(struct fdio_getset);

ret = ioctl(fildes, FDC_GET, &req);  /* Get current driver settings */
if (ret < 0) {
        perror("FDC_GET");
        exit(1);
};

getset.padcnt = 3;                   /* New padcnt is 3 bytes */
getset.TREQ = 100;                   /* New TREQ is 100 ms */
getset.rtmo = 10;                    /* New read time-out value is 10 sec */

ret = ioctl(fildes, FDC_SET, &cb);   /* Set new driver parameters */
if (ret < 0) {
        perror("FDC_SET");
        exit(1);
};
```

FILES

`/dev/fddi`*n*`/*`	FDDI interface special files
`/usr/include/sys/epackf.h`	
`/usr/include/sys/fd.h`	
`/usr/include/sys/fdsys.h`	
`/usr/include/sys/netdev.h`	

SEE ALSO

`close`(2), `ioctl`(2), `open`(2), `read`(2), `reada`(2), `write`(2), `writea`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

The ANSI documents for FDDI:

> *FDDI* `MAC` (Media Access Protocol) Specification (FDDI-MAC), document number X3.139-1987, November 5, 1986
>
> *FDDI* `PHY` (Physical Layer Protocol) Specification (FDDI-PHY), document number X3.148-1988, June 30, 1988
>
> *FDDI* `PMD` (Physical Medium Dependent) Specification (FDDI-PMD), document number X3.166-1990, September 28, 1989
>
> *FDDI* `SMT` (Station Management) Specification (FDDI-SMT), document number X3T9.5/84-49, Rev 7.2, June 25, 1992

Other documents related to FDDI:

> RFC 1390 Transmission of IP and ARP over FDDI Networks. January 1993. D. Katz
>
> Logical Link Control Specification (802.2 LLC), document number 802.2-1985, July 16, 1984

NAME

`fei` – Front-end interface

IMPLEMENTATION

Cray PVP systems systems

DESCRIPTION

The front-end interface (FEI) is a channel-to-channel adapter that connects Cray PVP systems to a front-end computer. The special file in `/dev` for the FEI is usually `/dev/fei`. For details on the special files in `/dev` at your site, see your system support staff. Currently, support for the FEI is provided through the I/O subsystem (IOS) as if the FEI were a Network Systems Corporation (NSC) adapter, except that the IOS driver provides only one logical connection or logical path. The device is otherwise treated as an NSC adapter; for details of usage, see `hy`(4).

FILES

`/dev/fei`

`/usr/include/sys/hy.h`

`/usr/include/sys/hysys.h`

SEE ALSO

`hy`(4), `vme`(4)

`ioctl`(2), `listio`(2), `read`(2), `reada`(2), `write`(2), `writea`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`fmsg` – GigaRing I/O message and MMR interface

IMPLEMENTATION

CRAY T90 systems with GigaRing-based I/O

CRAY J90 systems with GigaRing-based I/O

DESCRIPTION

The files in `/dev/fmsg` are character special files that allow sending and receiving of message and MMR (Mapped Memory Register) packets to GigaRing I/O nodes. Each file represents one GigaRing I/O node. The following F-transmission protocol packets are supported:

e packets	echo packets
g packets	MMR packets

User-level commands such as `fping`(8) and `mmr`(8) open `/dev/fmsg` devices and send and receive packets using read and write system calls.

The `fping`(8) command sends an echo packet to the specified GigaRing I/O node. The I/O node will then echo the packet back to the sender.

The `mmr`(8) command allows reading and writing of an I/O node's GigaRing MMR for ring management and error monitoring purposes.

The files in `/dev/fmsg` are normally created using the `mkfm`(8) command, based on the `mknod`(8) specifiecation detailed below.

The `mknod`(8) command for `/dev/fmsg` devices is as follows:

`mknod` *name type major minor reserved ionode*

name	Name of the `/dev/fmsg` file. Normally named for the I/O node ring and node address.
type	Devices in `/dev/fmsg` are character devices denoted by a `c`.
major	The major device number is `dev_fmsg`.
minor	Minor device number.
reserved	Must be 0.
ionode	The ring and node address of the target I/O node broken down in octal as follows:

0*rrrnn* where:

rrr	= I/O node ring number
nn	= I/O node node number

FILES

`/dev/fmsg/*`

`/usr/include/sys/fmsg.h`

`/usr/src/c1/io/fmsg.c`

SEE ALSO

`fping`(8), `mkfm`(8), `mmr`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`fslog` – File system error log interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `/dev/fslog` pseudo device is a read-only device that holds file system error log records. The file system error log daemon, `fslogd`(8), reads and processes those records to enable graceful handling of file system, directory, and inode errors detected by the kernel. For more information about the file system error log file, see `fslrec`(5).

FILES

`/dev/fslog`	Source of file system error log records

SEE ALSO

`fslrec`(5) for more information about the file system error log file

`fslogd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

hdd – HIPPI disk device interface

IMPLEMENTATION

Cray PVP systems with IOS model E

DESCRIPTION

The files in /dev/hdd are special files that allow read and write operations to HIPPI disk array devices. Each file represents one slice of a HIPPI disk device. The files in /dev/hdd are character special files that may be used directly to read and write HIPPI disk slices. Usually, they are called to perform I/O on behalf of higher-level logical disk device drivers. For I/O on a character disk device, read and write operations must transfer multiples of the HIPPI disk device sector size and all seek operations must be on HIPPI disk sector size boundaries.

The UNICOS operating system supports standard HIPPI disk array devices that adhere to the ANSI Standard IPI-3 command set on top of the ANSI Standard HIPPI Framing protocol.

The files in /dev/hdd are not mountable as file systems, although you may combine one or more HIPPI disk slices to make a mountable logical disk device (see dsk(4), ldd(4), and mount(8)). To create the files in /dev/hdd, use the mknod(8) command. Each must have a unique minor device number, along with other parameters used to define a HIPPI disk slice.

The mknod(8) command for HIPPI disk devices is as follows:

mknod *name type major minor dtype iopath start length flags reserved unit ifield*

name	Descriptive file name for the device (for example, hdd/scr0230.0).
type	Signifies how data will be transferred. Devices in /dev/hdd are character special devices denoted by a c.
major	Major device number for HIPPI disk devices. The dev_hdd name label in the /usr/src/uts/c1/cf/devsw.c file denotes the major device number for HIPPI disk devices. You can specify the major number as dev_hdd.
minor	Minor device number for this slice. The maximum number of HIPPI disk slices is defined in the deadstart parameter file by the HDDSLMAX parameter.
dtype	HIPPI disk device types are defined in /usr/src/uts/c1/sys/pddtypes.h. The HIPPI disk device types currently correspond to the HIPPI disk sector size.

```
HD16    1       /* 16k byte sector, iou = 4  */
HD32    2       /* 32k byte sector, iou = 8  */
HD64    3       /* 64k byte sector, iou = 16 */
```

The iou (I/O unit) is the size of the sector in multiples of 512-word (4096-byte) blocks.

iopath	On CRAY Y-MP systems with an IOS-E, the *iopath* specifies the I/O cluster, the I/O processor (IOP), and the controller channel number. For example, an *iopath* of 01234 is IOC 1, IOP 2, channel 34. A HIPPI channel pair takes up two IOP channels, the lower number channel for input and the upper channel number for output. The following is a typical HIPPI IOP configuration that has two pairs of HIPPI channels:

```
HIPPI 0         channel 30      input
                channel 32      output

HIPPI 1         channel 34      input
                channel 36      output
```

When specifying the HIPPI disk *iopath*, the input channel numbers are used (for example, HIPPI 1 on IOC 1, IOP 2, would have an *iopath* of 01234).

For information on setting the *iopath* field when configuring an `hdd` device node on CRAY EL and CRAY J90 systems, see the subsection "The iopath field."

start	Absolute starting sector number of the slice.
length	Number of blocks sectors in the slice.
flags	Flags for HIPPI disk device control, defined in `sys/hdd.h`, follow. They mainly are used for diagnostic and maintenance purposes. Usually, the flags field should be 0 for slices in `/dev/hdd`. The following flags are defined for `hdd` devices.

```
#define HS_CONTROL  0001   /* control device */
#define HS_NODEVINT 0002   /* no device intimate functions */
#define HS_MOVER    0004   /* this dev 3rd party data mover */
#define HS_NOMOVI   0010   /* no intermediate mover response */
#define HS_NOERREC  0040   /* no error recovery */
```

NOTE: If the HIPPI disk array is not a Cray Research supported network disk product, you may have to set the `HS_NODEVINT` flag.

For information on the `HS_DYNPATH` flag, see the subsection "The iopath field."

reserved	This field is reserved for future use.
unit	This field contains the HIPPI disk array unit number (also known as the *facility address*) and the raid partition number. The low-order 9 bits (bits 0 through 8) represent the facility address, *f*. Bits 9 through 15 represent the raid partition number, *r*: `0`*rrrfff*. To designate an octal value, specify the leading 0 on this parameter in the `mknod` command.
ifield	This is the HIPPI array *ifield* address if the array is connected through a HIPPI switch. Bit 2ˆ24 (camp on connect) is forced on by the driver.

The iopath field

On CRAY EL and CRAY J90 systems, you can use the `HS_DYNPATH` flag to create the `hdd` device node by using the `mknod`(8) command.

The `HS_DYNPATH` flag has a value of octal 0400. When the *flags* field in the `hdd` device node has the `HS_DYNPATH` bit set, the *iopath* is treated as a channel mask as opposed to a single channel. When using the channel mask, I/O to or from the `hdd` disk can occur over any of the channels specified in the channel mask.

When the `HS_DYNPATH` bit (octal 0400) is not set in the flags field, I/O to or from the `hdd` disk occurs over the single channel specified in the *iopath* field. There are seven possible input channel values for the *iopath* field for CRAY EL systems: 024, 040, 044, 060, 064, 0100, or 0104. There are 15 possible input channel values for the *iopath* field for CRAY J90 systems: 024, 030, 034, 040, 044, 050, 054, 060, 064, 070, 074, 0100, 0104, 0110, or 0114.

When the `HS_DYNPATH` bit (octal 0400) is set, the *iopath* field in the `hdd` device node represents a bit mask. On CRAY EL systems, this bit mask can be up to 7-bits wide; on CRAY J90 systems, it can be up to 15-bits wide. The bits in the bit mask in CRAY EL systems do not represent the same input channels as the bits in the bit mask in CRAY J90 systems.

The formats of the bit mask in the *iopath* field follow.

The *iopath* bit mask for CRAY Y-MP systems is of the following format (up to 7-bits wide):

Bit	Input channel
2ˆ0	024
2ˆ1	040
2ˆ2	044
2ˆ3	060
2ˆ4	064
2ˆ5	0100
2ˆ6	0104

Examples:

iopath bit mask (binary)	*iopath* bit mask (decimal)	Represents input channels
1000001	65	024, 0104
0111000	56	060, 064, 0100
0000011	3	024, 040
0101010	42	040, 060, 0100

iopath bit mask (binary)	*iopath* bit mask (decimal)	Represents input channels
1010101	85	024, 044, 064, 0104
1110111	119	024, 040, 044, 064, 0100, 0104

The *iopath* bit mask for CRAY J90 systems is of the following format (up to 15-bits wide):

Bit	Input channel
2ˆ0	024
2ˆ1	030
2ˆ2	034
2ˆ3	3040
2ˆ4	044
2ˆ5	050
2ˆ6	054
2ˆ7	060
2ˆ8	064
2ˆ9	070
2ˆ10	074
2ˆ11	0100
2ˆ12	0104
2ˆ13	0110
2ˆ14	0114

Examples:

iopath bit mask (binary)	*iopath* bit mask (decimal)	Represents input channels
000000001000001	65	024, 054
100000000111000	16440	040, 044, 050, 0114
101010101010101	21845	024, 034, 044, 054, 064, 074, 0104, 0114
010101010101010	10922	030, 040, 050, 060, 070, 0100, 0110

Facility addressing

The path to a HIPPI disk facility is defined by the I/O path, *ifield*, and unit number. The `HDDMAX` parameter describes the maximum number of HIPPI disk facilities in the `deadstart` parameter file.

Third-party transfer requests

The IEEE Mass Storage Reference Model breaks the process of doing I/O into its component parts and demonstrates the separation of data and control paths. As looked at from the peripheral's perspective, a control path handles functional request and response information, while the data mover path just moves data. This can allow for centralization of control and can better use high-bandwidth connections for moving and distributing data across a network.

The `hdd` driver has both data mover and data server capabilities. When acting as a data mover, an `hdd` connection is a slave to data transfers being controlled by a server path. When performing the server role, the `hdd` connection sends the read or write function to the device, along with the data path information needed to inform the data mover where to move the data.

To configure an `hdd` connection as a data mover, simply set the `HS_MOVER` flag, as defined previously, in the device inode. When a node is configured as a data mover, the command and response information travel a different path and originate from a server, possibly from a different host. A server functionality is the necessary complement to the data mover function.

Data move functionality for the `hdd` driver is provided with the UNICOS standard `reada`(2) and `writea`(2) system calls. Server functionality is provided through `ioctl` system calls to the `hdd` driver. A unique transfer identifier (tid) logically connects a data move operation with a server request.

`ioctl` requests

The format for `ioctl` requests that the `hdd` driver supports is as follows:

```
#include "sys/pddtypes.h"
ioctl( fildes, command, arg );
```

The `hdd` driver supports the following `ioctl` requests:

`HDI_READFOR` (0101)	Reads data to another host.
`HDI_WRITEFOR` (0102)	Writes data from another host.

The `HDI_READFOR` and `HDI_WRITEFOR` `ioctl` functions provide the `hdd` driver with a server capability; that is, read and write requests may be sent on behalf of data that is moved down another path to or from another host. You can use the `HDI_READFOR` and `HDI_WRITEFOR` commands in conjunction with an `hdd` node set up to be a data mover. The `hdi_serv` structure passes the appropriate information to the driver.

```
ioctl( fildes, HDI_READFOR, arg )
structure hdi_serve *arg;

/*
 *  Structure for hdd ioctl read/write for
 */
struct hdi_serv {
        uint    nblks   :32,  /* number of 512 word blocks */
                blkno   :32;  /* starting block in 512 word blks */
        uint    tid     :32,  /* transfer identifier */
                offset  :32;  /* byte offset in destination buffer */
        uint    ifield  :32,  /* destination ifield */
                port    :32;  /* destination controller port */
};
```

HDI_CANCEL (0103) Cancels a request to a data mover.

```
int tid;
ioctl( fildes, HDI_CANCEL, tid )
```

The HDI_CANCEL ioctl command cancels a pending data move identified by the transfer identifier (tid).

HDI_GET_TID (0104) Gets a unique transfer identifier (tid).

```
int tid;
ioctl( fildes, HDI_GET_TID, &tid )
```

Gets a system-unique nonzero transfer identifier (tid) that may be used to identify a data mover and/or data server request.

HDI_SET_MOV_TO (0105) Sets data mover request time-out.

```
int timeout = 60;
ioctl( fildes, HDI_SET_MOV_TO, &timeout );
```

Sets the time-out value for a data move to the specified number of seconds. When the timer for a pending data move expires, the pending move is canceled and the data move request is terminated with an EINTR errno.

EXAMPLES

The following mknod command makes a node for hdd/scr0334.2, type c, major number dev_hdd, minor number 110, disk type HD04, I/O cluster 0, IOP 3, channel 34, starting at block 0, length of 100,000 blocks, 0 for flags, facility address of 020, raid partition number 021, and an *ifield* address of 7:

```
mknod hdd/scr0334.2 c dev_hdd 110 1 0334 0 100000 0 0 021020 7
```

FILES

`/dev/hdd/*`

`/usr/include/sys/epackj.h`

`/usr/include/sys/hdd.h`

`/usr/include/sys/pddprof.h`

`/usr/include/sys/pddtypes.h`

`/usr/src/c1/io/hdd.c`

SEE ALSO

`dsk`(4), `ldd`(4), `mdd`(4), `pdd`(4), `xdd`(4), `sdd`(4), `ssdd`(4)

`ddstat`(8), `hddmon`(8), `mknod`(8), `mount`(8), `sdstat`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`HIPPI` – ANSI High Performance Parallel Interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `HIPPI` interface (or `HIPPI` driver) drives an ANSI standard High Performance Parallel Interface (HIPPI) channel. The driver supports no device in particular. If two machines are directly connected by using HIPPI channels, the driver behaves in a manner similar to UNICOS named pipes. If the HIPPI channel is connected to a device, user software must execute the device's protocol. Application processes use the `HIPPI` driver by means of the standard UNICOS system calls: `close`(2), `ioctl`(2), `listio`(2), `open`(2), `read`(2), `reada`(2), `write`(2), and `writea`(2). Each of the special files in a `/dev/hippi*` directory represents one input or output HIPPI channel.

By convention, HIPPI file names have the following format:

`/dev/hippi`*n*`/i`*xx*
`/dev/hippi`*n*`/o`*xx*

n	Physical channel number
`i`	Input channel
`o`	Output channel
xx	Logical channel number

For operation in one direction only, only one channel must be opened. If the application uses both read and write operations, it must open both an input and an output channel. A read operation is valid only on an input channel; a write operation is valid only on an output channel. All HIPPI I/O is raw; the user process is locked in memory as data moves directly between the user buffer and the channel. Therefore, user buffers must be word-aligned, and their length must be a multiple of 8 bytes.

HIPPI Fundamentals

HIPPI is a 32-bit parallel unidirectional point-to-point data channel. Usually, it is installed in input/output pairs for bidirectional operation. Data flows from a HIPPI *source* to a *destination*.

From the user's perspective, the basic unit of information on the channel is a *packet*. The channel hardware breaks up packets into *bursts* of 256 32-bit words. READY pulses from the destination to the source control flow; each READY pulse lets the source send one burst. The PACKET signal from source to destination marks the boundaries between packets.

There is a signal called `REQUEST` from source to destination, and a companion signal called `CONNECT` in the opposite direction. When both signals are present, a *connection* is said to exist. Data may not flow unless there is a connection.

To establish a connection, the source raises the `REQUEST` signal to the destination. At this time, it can place 32 bits of information on the data lines, called the *I-field*. The destination can examine the I-field before it responds to the request. The destination responds either by accepting the request or rejecting it. Acceptance consists of raising the `CONNECT` signal and transmitting READY pulses. To reject a request, the destination raises `CONNECT` for a certain period of time and drops it again without transmitting any `READY` pulses. The source responds by dropping `REQUEST`.

Connections may be broken either by the source (by dropping the `REQUEST` signal) or by the destination (by dropping `CONNECT`). The other side must respond by dropping its corresponding signal. The source must cease sending data when the connection is broken. When a destination breaks a connection, data in transit can be lost.

For further information about the `HIPPI` interface, see the ANSI documents listed in the SEE ALSO section.

Dedicated and Shared HIPPI Channels

The `HIPPI` driver supports two modes of operation: dedicated and shared. The mode is assigned to each channel when it is opened. If a channel is dedicated, only one process can have that channel open at a time. If a channel is shared, several processes can use one HIPPI channel. The driver enforces a protocol on the shared channel, allowing it to determine the destination of each incoming message.

Shared channels are not supported on Cray PVP systems configured to read and write the HIPPI channels with SSD solid-state storage buffers.

For more information on configuring HIPPI channels on CRAY Y-MP systems, see *UNICOS Networking Facilities Administrator's Guide*, Cray Research publication SG–2304.

HIPPI `ioctl` Requests

Several `ioctl` requests are available. They have the following format:

```
#include <sys/hx.h>
ioctl (fildes, command, arg)
struct hxio *arg;
```

The valid `ioctl` requests are as follows:

`HXC_GET` Gets the current driver parameter settings; stores them in the `hxio` structure referenced by *arg*.

`HXC_SET` Sets driver parameters from the structure referenced by *arg*.

Connections are established automatically in response to `read`(2) and `write`(2) system calls. To control the termination of connections, use the `HXCF_DISC ioctl` flag (see the following).

The `hxio` structure contains the following fields:

`unsigned int flags` Flags that control channel operation. The *flags* field controls the driver's options. The bits in the flags field are defined as follows:

`HXCF_DED`	The channel is open in dedicated mode; no other processes may share the channel. If this flag is set, the value in the *path* field is meaningless. The `ioctl` request `HXC_GET` is used to set this flag. The `ioctl` request `HXC_SET` ignores this flag.
`HXCF_DISC`	(Meaningful only on output devices) When set, causes the driver to end the HIPPI connection after each packet written.
`HXCF_DOWN`	If this bit is set, the channel is unavailable to other users. If this bit is clear, the channel is available. When the channel is down, the driver returns the `ENXIO` error. Programs that reconfigure the channel should issue a `close` request immediately after the `ioctl` request, then reopen the channel before trying to do anything else. This bit is set in the `ioctl` request `HXC_SET` on a dedicated channel. This is a read-only flag.
`HXCF_HDR`	Controls the handling of the HIPPI-FP header (first word of each packet). If `HXCF_HDR` is set (default), the HIPPI-FP header is assumed to be in the user buffer. If clear, the `HIPPI` driver adds a HIPPI-FP header to the beginning of the user data on output and strips it off on input. For HIPPI-FP field definitions, see `sys/hippifp.h`. When creating the HIPPI-FP header, the driver sets the `ulpid` field to the value of `path` and the `d2size` field to the user buffer length (in bytes). All other HIPPI-FP fields are 0. The `HIPPI` driver does not validate the HIPPI-FP header on input.
`HXCF_HIPPI`	(Read-only, not user settable) Indicates channel is a HIPPI. Clear for HSX.
`HXCF_IND`	Controls the passing of the I-field value to the driver on output channels. By default, the driver uses the value in `ifv`. If `HXCF_IND` is set, the driver uses the low-order 32 bits of the first word of the user buffer as the I-field. If `HXCF_HDR` is set, the HIPPI-FP header is in the second word of the user buffer. `HXCF_IND` is convenient if the application must change the I-field between packets, especially if `writea`(2) is being used.
`HXCF_IO`	(Not user-settable) Indicates the channel direction: set for output, and clear for input.

`HXCF_ISB`	Controls the sending of an Initial Short Burst. The HIPPI-PH specification allows either the first or last burst of a packet to consist of less than 256 HIPPI words (32- or 64-bit words). The `HIPPI` driver sends short bursts last by default. To send it, the user must first set this flag and use a `listio`(2) request with two output buffers, and the `HXLI_CHD` flag set for only the first one. The first buffer contains the contents of the first burst, and it may consist of from 1 to 256 Cray words. The second buffer must be a multiple of 128 Cray words for 32-bit HIPPI channels and 256 Cray words for 64-bit HIPPI channels.
`HXCF_MODEL_E`	(Not user-settable) Indicates the type of IOS: set for IOS-E and CRAY EL memory HIPPI systems, and clear for others.

`int err` Detailed error status from the last request; for a list of HIPPI errors, see the Messages section.

`int path` ULP-ID for shared input channels or for autoheader mode for output channels.

`unsigned int tmo` Time-out value (in seconds).

`int ifv` (I-field value) The output channel driver sends this value when it requests a connection. The input channel driver uses this value to decide whether to accept or reject a connection request. The input driver forms the logical product of the channel's I-field value and the user's I-field mask. If this product matches the user's I-field value, the driver accepts the connection. If the values do not match, the driver rejects the connection.

On CRAY EL systems, if the input driver does not use the `ifv` field, all incoming connections are accepted after the input channel has been opened.

`int ifm` (I-field mask) The input driver makes a logical product with this mask before comparing the I-field on the channel with the I-field value. The `ifm` field is not used on CRAY EL systems.

`int ctmo` Connection time-out value (in seconds); determines how long the driver will wait for a connection to become established after it is requested.

`listio` Special Features

The `HIPPI` driver defines one flag to be used in the `li_drvr` field of the `listreq` structure (see `listio`(2)), as follows:

`HXLI_CHD` Indicates, in effect, that user data is chained. When this flag is set, the driver suppresses the end-of-block signal at the end of the current list entry (for an output channel) or allows the current input data block to overflow into the next list entry (for an input channel).

If the driver encounters an end-of-block signal during a read operation in which the `HXLI_CHD` flag is set, no error indication is returned to the user. Subsequent read operations will complete with a data length of 0 until the `HXLI_CHD` flag is cleared. If the user issues the `ioctl` request `HXC_GET` to check the status after a chained read operation, the `err` field of the `hxio` structure is set to the `HXST_EOB` error code, indicating that an end-of-block signal arrived before the last read request was processed.

If the driver has not detected an end-of-block signal on a unchained read operation (that is, `read`(2), `reada`(2), or `listio`(2) with the `HXLI_CHD` flag clear) by the time the read operation completes, the driver discards the unread part of the block. The byte count returned to the user is equal to the size of the buffer, and no error code is returned in `errno`. If the user issues the `ioctl` request `HXC_GET` to check the status after a unchained read operation, the `err` field of the `hxio` structure is set to the `HXST_LONG` error code, indicating that the entire block was not read and the remainder was discarded.

When using the `listio` chaining feature on shared channels to combine more than one buffer to make a single data block, the user must include all buffers in the same `listio`(2) system call. That is, make sure that the `HXLI_CHD` flag is clear in the `li_drvr` field of the last item for an HIPPI channel in each `listio`(2) call. Failure to do this can cause lost or corrupt data on the channel if the user process is swapped between requests.

The IOS-E systems support chained buffers only in pairs; that is, if `HXLI_CHD` is set for one buffer, it must be clear for the next.

On CRAY EL systems with memory HIPPI, the chained buffers also must be supplied in pairs. The first buffer also must consist of 1024 bytes or less. On the input channel, the first burst will be written to the first buffer. If the first burst fits completely into the first buffer, the remainder of the packet will be written to the second buffer, leaving empty space in the first buffer if the first burst did not fill it completely. If the first burst is bigger than the first buffer, the first buffer will be filled, the remainder of the first burst will be written to the second buffer, and the remainder of the packet will be appended to the second buffer.

HIPPI Protocol

On dedicated channels, the `HIPPI` driver treats all HIPPI packets as opaque data: no specific protocol is required or recognized.

Shared channel operation requires that all applications use the ANSI draft HIPPI Framing Protocol (HIPPI-FP). The `sys/hippifp.h` header file contains definitions for the header. The `HIPPI` driver examines the ULP-ID field in each input packet. For an application to receive a packet in shared mode, its *path* value must match the contents of this field. The default value of *path* is *n*+128; *n* = (minor device number) modulo 16 (for example, the default path (ULP-ID) for `/dev/hippi/i01` and `/dev/hippi/o01` is 129; for `/dev/hippi/*02` is 130, and so on). Each application can change its ULP-ID by setting the new value in the *path* variable in a `HXC_SET` request.

HIPPI 800- and 1600-M Modes

The HIPPI-PH specification describes both 32-bit and 64-bit HIPPI implementations. Nominal data rates are 800 M and 1600 M, respectively. One cable in each direction, called Cable A, is used for 800-M HIPPI. 1600-M HIPPI uses two cables in each direction, called Cable A and Cable B. Cray Research systems that have an IOS-E can be wired for Cable B; all other Cray Research systems have only 32-bit HIPPI.

The ANSI HIPPI-SC (Switch Control) draft standard allocates bit 2**28 of the I-field to control the mode of switch connections. The Cray Research `HIPPI` driver uses this same bit to select the channel mode, as follows:

Output	If bit 2**28 of the I-field is set and the output HIPPI has a second cable installed and connected, the driver selects 64-bit mode for the transfer. If there is no second cable, the driver returns `EIO`.
Input	The driver examines bit 2**28 of incoming I-fields. If set, and Cable B is installed and connected, the driver selects 64-bit mode for the transfer.

MESSAGES

When a fatal error occurs, the `HIPPI` driver returns one of the following error codes in `errno`. The error codes and their meanings are as follows:

`EFAULT` A bad argument address was specified in an `ioctl` request.

`EINVAL` The driver software has detected a fatal parameter error. Errors in system configuration and user errors in system call invocation, can cause this error.

`EIO` One of the following conditions can cause this error.

- A fatal I/O error occurred or the HIPPI channel closed while asynchronous I/O was active (on a `read`(2) or `write`(2) system call).
- A data block was too long for the input buffer (on a `read`(2) system call) or overflowed the channel (on a `write`(2) system call).
- An I/O request timed out (on a `read`(2) or `write`(2) system call).

The detailed error status is available in the `err` field of the `hxio` structure; to see this field, use the `ioctl` request `HXC_GET`.

`ELATE` An input request timed out.

`ENXIO` The HIPPI channel is unavailable (on an `open`(2) system call), or the channel is not open (on a `close`(2) system call). On Cray PVP systems, this code can mean that the IOP failed to allocate some resource.

After a fatal error, the detailed error status is available with the `ioctl` request `HXC_GET`. The driver returns error codes in the `err` field of the `hxio` structure.

The following detailed error codes are defined on CRAY EL systems that have VME HIPPI:

`HXST_BUF`	IOS I/O buffer unavailable on open operation.
`HXST_CHAN`	CPU gave bad channel number to IOS; caused by configuration error.
`HXST_DBG`	Debug mode error; indicates a driver fault (should never occur).
`HXST_EOB`	Unexpected end of packet (input only).
`HXST_FLGS`	Buffer flags do not match; indicates a driver fault (should never occur).

`HXST_FMEM`	IOS free memory unavailable on open operation.
`HXST_FNC`	Illegal function code; indicates a driver fault (should never occur).
`HXST_HISP`	No high-speed channel to this IOP; caused by configuration error or wrong target memory.
`HXST_LLEN`	Transfer length too long; indicates a driver fault (should never occur).
`HXST_LONG`	Long block received (input only).
`HXST_MOS`	MOS buffer unavailable on open operation (debug mode only).
`HXST_NDEV`	No device present on the channel (hardware signal).
`HXST_OK`	No error (binary 0).
`HXST_OPEN`	Channel is not open; indicates a driver fault (should never occur).
`HXST_OVER`	Data overrun error (input only).
`HXST_TM`	Bad target memory type; indicates a driver fault (should never occur).
`HXST_TMO`	Request timed out.
`HXST_ZLEN`	Buffer length is 0; indicates a driver fault (should never occur).
`HXST_CTMO`	Connection timed out.
`HXST_CNPR`	Connection not present.
`HXST_CREJ`	Connection rejected.
`HXST_DISC`	Channel disconnected.
`HXST_CNPE`	No connection pending.
`HXST_CAPR`	Connection already present.
`HXST_CABT`	Connection aborted.
`HXST_LLRC`	Length/Longitudinal Redundancy Checkword (LLRC) error.
`HXST_CPE`	Channel parity error.
`HXST_CHST`	Channel status is not valid.
`HXST_BOE`	Buffer overrun error.
`HXST_OIM`	Odd initial microburst.
`HXST_BPE`	Buffer parity error.
`HXST_IPE`	I-field parity error.

The following detailed error codes are defined for Cray Research systems that have an IOS-E and CRAY EL systems that have VME HIPPI:

`HIST_INV_REQ`	Request code is not valid.
`HIST_INV_RL`	Request packet length is not valid.

`HIST_IS_UP`	Channel already configured up.
`HIST_NOT_UP`	Channel is not configured up.
`HIST_NOT_IMP`	Function is not implemented.
`HIST_RTMO`	Read request time-out.
`HIST_DRV_DOWN`	Driver terminated by configuration down.
`HIST_IS_OPN`	Logical path already open.
`HIST_RAW_OPN`	Bad raw channel open request.
`HIST_NOT_OPN`	Logical path is not open.
`HIST_PTH_CLS`	Read abandoned because path is closed.
`HIST_LONG`	Long packet excess discarded (nonfatal).
`HIST_DTA_ERR`	Data integrity error on read.
`HIST_CHAN_TMO`	Channel activation time-out.
`HIST_DRV_TERM`	Driver terminating.
`HIST_NOT_CNCT`	Interconnect-A not present.
`HIST_HALT_IO`	Read/write request returned due to halt-io.
`HIST_INV_PARM`	Configure `UP` parameter that is not valid.
`HIST_NOT_64`	Cannot write in 64-bit HIPPI mode.
`HIST_CN_REJ`	Connection attempt was rejected.
`HIST_CN_FAIL`	Connection attempt failed (other).
`HIST_CN_TMO`	Connection request time-out.
`HIST_CN_STUCK`	Connect-in will not drop.
`HIST_INV_SF`	Device control subfunction that is not valid.
`HIST_HANGUP`	Connection went away.
`HIST_SECDED`	SECDED error in I/O buffer.
`HIST_INTRCNCT`	Lost Interconnect.
`HIST_EOB`	Short block received.

The following detailed error codes are defined for IOS-E systems:

`HIST_RC_OK`	No errors
`HIST_RC_PARAM`	Parameter error
`HIST_RC_NO_EVENT`	Requested event not outstanding
`HIST_RC_NOT_QUED`	Entry was not on specified queue

`HIST_RC_Q_EMPTY`	Queue is empty
`HIST_RC_BAD_CB`	Invalid I/O buffer ordinal
`HIST_RC_BAD_ADR`	Invalid I/O buffer address
`HIST_RC_INVDIR`	Invalid transfer direction
`HIST_RC_TMNAVL`	Target memory channel not available/configured
`HIST_RC_HWERR`	Unrecovered hardware error
`HIST_RC_QUEUED`	Entry already on a queue
`HIST_RC_NOMEM`	Memory space not available
`HIST_RC_BADLEN`	Bad memory allocation length
`HIST_RC_BADREQPKT`	Bad request packet address on release
`HIST_RC_BADRESPKT`	Bad respond packet address
`HIST_RC_QUEREQ`	Must queue request (can't set pointer)
`HIST_RC_CBNAVL`	Channel buffer not available to reserve
`HIST_RC_CBNOTOWN`	Channel buffer not owned by subsystem
`HIST_RC_BADSSID`	Invalid subsystem identifier

EXAMPLES

The following is an example of the usage of the `hxio` structure to alter driver defaults on CRAY Y-MP systems:

```
#include <sys/hx.h>
struct hxio cb; /* HIPPI driver parameter structure            */
int s;          /* Status returned from ioctl() request        */
int fildes;     /* HIPPI output device file descriptor         */

s = ioctl(fildes, HXC_GET, &cb);  /* Get current driver settings  */
if(s  < 0)      {
        perror("HXC_GET");
        exit(1);
}
cb.flags |= HXCF_DISC;  /* Automatic disconnect after each output */
cb.ifv = ifield;        /* I-field value for all packets          */
         /* OR */
cb.flags |= HXCF_IND;   /* I-field prefixed to data in buffer     */
cb.path = 0300;         /* Set new ULP-ID value                   */
cb.ctmo = 10;           /* Connection timeout value is 10 seconds */
s = ioctl(fildes, HXC_SET, &cb);  /* Set new driver parameters    */
if(s  < 0)      {
        perror("HXC_SET");
        exit(1);
}
```

FILES

/dev/hippi*/*	HIPPI channel special files
/usr/include/sys/hx.h	
/usr/include/sys/hpacket.h	(CRAY Y-MP systems and CRAY EL systems with VME)
/usr/include/sys/hxsys.h	(CRAY Y-MP systems and CRAY EL systems with VME)

SEE ALSO

`hsx`(4)

`close`(2), `ioctl`(2), `listio`(2), `read`(2), `reada`(2), `write`(2), `writea`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`hsxconfig`(8), `mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

The ANSI documents for `HIPPI`:

HIPPI Mechanical, Electrical, and Signaling Protocol Specification (HIPPI-PH), document number X3T9.3/88-127, Rev 8.1, June 24, 1991

HIPPI Framing Protocol (HIPPI-FP), document number x3T9.3/89-146, Rev 4.2, June 24, 1991

HIPPI 802.2 Link Encapsulation (HIPPI-LE), document number X3T9.3/90-119, Rev 3.1, June 28, 1991

HIPPI Physical Switch Control (HIPPI-SC), document number X3T9.3/91-023, Rev 1.9, June 28, 1991

NAME

`hpi3` – IPI-3/HIPPI packet driver configuration file

IMPLEMENTATION

Cray PVP systems that have an IOS model E

DESCRIPTION

The IPI-3/HIPPI packet driver configuration file consists of statements that describe the I/O processors (IOPs), channels, slaves, and devices that compose the IPI-3/HIPPI subsystem. It also includes a list of driver options and limits that, when specified, override the system defaults.

This man page describes the format of the configuration file and the IPI-3/HIPPI packet driver `ioctl` requests. For information on the IPI-3/HIPPI packet driver commands, see the `hpi3_clear`(8), `hpi3_config`(8), `hpi3_option`(8), `hpi3_start`(8), `hpi3_stat`(8), and `hpi3_stop`(8) man pages in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022.

Configuration File Format

Each line in the configuration file is a configuration type statement, a configuration description statement, a comment, or white space.

To specify comments, begin a line with the # symbol. Configuration type statements begin with the `-` symbol, followed by one of these strings: `IOPS`, `CHANNELS`, `SLAVES`, `DEVICES`, or `OPTIONS`. You must specify the configuration type statements in the order listed. All lines specified between configuration type statements that are not comments or white space are configuration description statements. The format of the configuration description statements depends on the configuration type statement preceding it.

The configuration file format is as follows:

`-IOPS`
iop description statements (CRAY Y-MP systems with IOS model E (IOS-E) only)

`-CHANNELS`
channel description statements

`-SLAVES`
slave description statements

`-DEVICES`
device description statements

`-OPTIONS`
option description statements

IOP Description

(CRAY Y-MP systems with an IOS-E only) The IOP description statements follow the `-IOPS` statement. Each IOP description statement describes one IOP. You can specify 32 IOP description statements. You cannot specify an IOP description for CRAY EL series and CRAY J90 series systems. The description statement must be in the following format:

iop-name cluster-number iop-number

iop-name	Consists of 1 to 8 characters and must be unique. The name is used in creating a file that can be used to issue packets that affect the IOP as a whole, rather than device-specific requests.
cluster-number	Must be in the range 0 to 15.
iop-number	Must be in the range 0 to 3. Each cluster-IOP pair must be unique.

Channel Description

The channel description statements follow the `-CHANNELS` statement. Each channel statement describes one channel. The description statement must be in the following format for CRAY Y-MP system with IOS-E:

iop-name input-channel output-channel state input-ch-timeout output-ch-timeout connect-timeout

The description statement must be in the following format for CRAY J90 and CRAY EL systems:

input-channel output-channel state input-ch-timeout output-ch-timeout connect-timeout

iop-name	(CRAY Y-MP systems with an IOS-E only) Denotes the name of the IOP in which the channel is attached. The IOP must already be defined in one of the IOP description statements.
input-channel	Specifies the channel number of the input subchannel. The subchannel is attached to EIOP *iop-name* on CRAY Y-MP systems with an IOS-E. For a list of valid channel pairs for CRAY J90 and CRAY EL systems, see the Channel Selection (CRAY J90 and CRAY EL systems) subsection.
output-channel	Specifies the channel number of the output subchannel. On CRAY Y-MP systems with an IOS-E, the subchannel is attached to EIOP *iop-name*, the channel numbers must be 030, 032, 034, or 036, and a channel number must be unique within an EIOP. For a list of valid channel pairs for CRAY J90 and CRAY EL systems, see the Channel Selection (CRAY J90 and CRAY EL systems) subsection.
state	Specifies the status (`UP` or `DOWN`) of the channel after starting the packet driver. If you specify `UP`, the channel pair will be configured up as part of the start-up sequence. If you specify `DOWN`, the channel pair is left down after the start-up sequence completes.
input-ch-timeout	Specifies the time-out period (in milliseconds) for requests issued to the input channel. The time-out period must be in the range 0 to 0177777. If you specify 0, the default time-out period is 10 seconds.

output-ch-timeout	Specifies the time-out period (in milliseconds) for requests issued to the output channel. The time-out period must be in the range 0 to 0177777. If you specify 0, the default time-out period is 10 seconds.
connect-timeout	Specifies the time-out period (in hundredths of a second) for the connection request. The time-out period and must be in the range 0 to 0177777. If you specify 0, the default time-out period is 10 seconds.

On CRAY Y-MP systems with an IOS -E, you can specify a maximum of two channel descriptor statements per IOP. CRAY EL systems can have up to 7 memory-HIPPI channels, and CRAY J90 systems can have up to 15 memory-HIPPI channels; for a list of valid channel pairs for CRAY J90 and CRAY EL systems, see the Channel Selection (CRAY J90 and CRAY EL systems) subsection.

Slave Description

The slave description statements follow the `-SLAVES` statement. Each slave statement describes one slave. The description statement must be in the following format for CRAY Y-MP systems with an IOS-E:

slave-name iop-name channel-pairs(s) i-field

The description statement must be in the following format for CRAY J90 and CRAY EL systems:

slave-name channel-pairs(s) i-field

slave-name	Denotes the name of the slave (attached to IOP *iop-name* on CRAY Y-MP systems with an IOS-E). A slave name must consist of 1 to 8 characters and must be unique. The slave name is used in the device definition to identify the paths to the device.
iop-name	(CRAY Y-MP systems with an IOS-E) Denotes the name of the IOP in which the slave is attached. The IOP must already be defined in one of the IOP description statements.
channel-pair(s)	Specifies the channel pairs that may be used to issue requests to the slave and to transfer data to the slave. The channel pairs must be in the following format:

input-channel1:output-channel1[`,` *input-channel2:output-channel2*]

If you specify two channel pairs, the channel pairs must be unique. The channel pairs specified must already be defined in a channel description statement. See the Channel Selection (CRAY J90 and CRAY EL systems) subsection for a list of valid channel pairs for CRAY J90 and CRAY EL systems.

i-field	The HIPPI i-field address. It is used to route requests and data from a HIPPI switch to a slave.

On CRAY Y-MP systems with an IOS-E, you may specify a maximum of 32 slaves per IOP.

Device Description

The device description statements follow the `-DEVICES` statement. Each device statement describes one device. The description statement must be in the following format:

device-name slave-name low-facility-address high-facility-address

device-name	Specifies the name of a device attached to slave *slave-name*. The device name must consist of 1 to 8 characters and must be unique. The name is used to create a file that can be used to issue packets to the device.
slave-name	Denotes the name of the slave in which the device is attached. The slave must have been defined in one of the slave description statements.

Facility addresses are used to route requests or data from a slave to a device. *low-facility-address* and *high-facility-address* specify a range of facility addresses. Asynchronous responses with facility addresses within this range are associated with the device. The facility addresses specified must be in the range 0 to 0xFF.

You may specify a maximum of 32 devices.

Option Description

The option description statements follow the `-OPTION` statement. Each option statement is in the format:

option option-value

Valid options are `IOS_OPTIONS`, `MAX_ASYNC`, `MAX_IOP_PROC`, `MAX_NON_CMDLST`, `MAX_STK_COUNT`, and `TRACING`.

`IOS_OPTIONS`	Defines a value used to control temporary and installation-specific IOP configuration options. If you omit this option, the IOS option value defaults to 0.
`MAX_ASYNC`	Defines the maximum number of asynchronous responses that may be enabled for an individual device. The number of asynchronous responses must be in the range 0 to 20. If you omit this option, the packet driver will default to a maximum of five asynchronous responses.
`MAX_IOP_PROC`	Defines the maximum number of processes that can open an IOP device concurrently. The number of processes must be in the range 1 to 50. If you omit this option, the packet driver will default to a maximum of 10.
`MAX_NON_CMDLST`	Defines the maximum number of noncommand list requests that may be stacked for an individual device. The stack count must be in the range 0 to 10. If you omit this option, the packet driver will default to a maximum of five.
`MAX_STK_COUNT`	Defines the maximum number of command list requests that may be stacked for an individual device. The stack count must be in the range 0 to 20. If you omit this option, the packet driver will default to a maximum of five.

`TRACING`	Specifies whether packet driver tracing should be on or off. The option value must be either `ON` or `OFF`.

Channel Selection (CRAY J90 and CRAY EL systems)

The following table indicates valid channel pairs for CRAY J90 systems:

Channel pair	Input channel	Output channel
1	024	027
2	030	033
3	034	037
4	040	043
5	044	047
6	050	053
7	054	057
8	060	063
9	064	067
10	070	073
11	074	077
12	0100	0103
13	0104	0107
14	0110	0113
15	0114	0117

The following table indicates valid channel pairs for CRAY EL systems:

Channel pair	Input channel	Output channel
1	024	027
2	040	043
3	044	047
4	060	063
5	064	067
6	0100	0103
7	0104	0107

Sample Configuration Files

The following is an example of an IPI-3/HIPPI packet driver configuration file.

```
#
#   IPI-3/HIPPI Packet Driver Configuration File
#

#
#   Define the IOPs.
#

-IOPS

#
#IOP Name     Cluster   IOP
#
iop_0_0          0       0
iop_0_2          0       2
iop_3_2          3       2
iop_3_3          3       3

#
#   Define the channels
#

-CHANNELS

#
#IOP Name
#      +Input Channel
#              +Output Channel
#                          +State
#                           (UP,DOWN)
#                                  +Input Channel
#                                   Timeout
#                                   (sec/1000)
#                                             +Output Channel
#                                              Timeout
#                                              (sec/1000)
#                                                          +Connection
#                                                           Timeout
#                                                           (sec/1000)

iop_0_0  030     032         UP      3000       3000        2000
iop_0_0  034     036         UP      3000       3000        2000
iop_0_2  030     032         DOWN    1000       1000        2000
iop_3_2  034     036         UP      2000       2000        3000
iop_3_3  034     036         DOWN    2000       3000        2000
```

```
iop_3_3  030     032         UP      3000       2000        3000

#
#   Define the slaves
#

-SLAVES

#
#Slave Name   IOP NAME    Channel Pair(s)   I-Field
#

slv_0         iop_0_0     030:032/034:036   00x01000007

slv_1         iop_0_2     030:032           00x01000007
slv_2         iop_0_2     030:032           00x01000008
slv_3         iop_0_2     030:032           00x01000009
slv_4         iop_0_2     030:032           00x0100000a

slv_5         iop_3_2     030:036           00x01000003

slv_6         iop_3_3     030:032           00x01000004
slv_7         iop_3_3     030:032           00x01000005
slv_8         iop_3_3     030:036           00x01000007

#
#   Define the devices
#

-DEVICES

#Device    Slave      Low Facility  High Facility
#Name                 Address       Address
#

er91       slv_0      0x01          0x01
er92       slv_0      0x02          0x02
er93       slv_0      0x03          0x03
er94       slv_0      0x04          0x04

dsk_1      slv_1      0xFF          0xFF
dsk_2      slv_2      0xFF          0xFF
dsk_3      slv_3      0xFF          0xFF
dsk_4      slv_4      0xFF          0xFF

hpdsk      slv_5      0xFF          0xFF

d1_1       slv_6      0xFF          0xFF
```

```
d1_2        slv_7      0xFF          0xFF
hpdsk_2     slv_8      0xFF          0xFF
```

The following is an example of a configuration file for a CRAY EL system:

```
################################################################################
#                                                                              #
#     Tape K-packet driver configuration file for CRAYELS systems              #
#                                                                              #
################################################################################
#
#    IPI-3 Tape Configuration
#

-CHANNELS

# +Input Channel
# |   +Output Channel
# |   |
# |   |     +State (UP,DOWN)
# |   |     |     +Input Channel
# |   |     |     |  timeout  +Output Channel
# |   |     |     |  (sec/1000)     |  timeout  +Connection
# |   |     |     |           |  (sec/1000)     |  timeout
# v   v     v     v           v           v  (sec/100)

0104  0107  UP    30000       30000       2000


-SLAVES
#
# +Tape Slave Name
# |         +Channel Pair(s)
# |         |
# |         |                             +I-Field
# |         |                             |
# v         v                             v

s_hippi1    0104:0107                     0x0100001d


-DEVICES
#
# +Tape Device Name (for /dev/hpi3 )
# |              +Slave name
# |              |           +Low Facility Address
# |              |           |     +High Facility Address
# |              |            |     |
# |              |            |     |
# |            |        |     |
# |              |            |     |
# v              v            v     v

hdd_dsk1     s_hippi1   0xFF  0xFF


-OPTIONS

#   Define the maximum number of command list requests
#   that may be stacked for an individual device

MAX_STK_COUNT           5


#   Define the maximum number of non-command list requests
#   that may be stacked for an individual device

MAX_NON_CMDLST           3


#   Define the maximum number of asynchronous responses
#   that may be enabled for an individual device

MAX_ASYNC         5


#   Define the maximum number of processes that can
#   open an iop device

MAX_IOP_PROC           10


#   Request kernel tracing to be on or off.

TRACING          On


#   IOS_OPTIONS defines a value used to control
#   temporary and installation-specific configuration options.

# IOS_OPTIONS     0xabc
```

`ioctl` Requests

You can use the following `ioctl`(2) requests with the IPI-3/HIPPI packet driver: `PKI_DRIVER_STS`, `PKI_ENABLE`, `PKI_GET_CONFIG`, `PKI_GET_DEVCONF`, `PKI_GET_DEVTBL`, `PKI_GET_OPTIONS`, `PKI_RECEIVE`, `PKI_SEND`, and `PKI_SIGNO`. A description of each of these requests follows, along with an example of its use.

PKI_DRIVER_STS

The PKI_DRIVER_STS request returns the status of the IPI-3/HIPPI packet driver. If the packet driver has been started, a negative value is returned. If the packet driver is down, 0 is returned.

You can issue this request to the request device, IOP devices, or IPI-3 devices.

The following example shows how to issue a request for the driver status:

```
/*
 *    Get the IPI-3/HIPPI Packet Driver Status
 */

#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/cnthpi3.h>
#include <sys/pki_ctl.h>

main()
{
           int         fd;

           /*
            *     Open the request device
            */

           if ( (fd = open(HPI3_REQ, O_RDWR ) ) < 0 ) {
              perror( "Unable to open the request device" );
              exit(errno);
           }

           if ( ioctl( fd, PKI_DRIVER_STS, 0 ) < 0 ) {
                    printf( "The packet driver has been started");
           } else {
                    printf( "The packet driver is not active" );
           }

           close(fd);
}
```

PKI_ENABLE

The PKI_ENABLE request enables the packet interface for a device. The driver allocates buffer space for the packets issued to the driver. You must issue this request before trying to register for a signal and before trying to send request packets.

You can issue this request only to IOP devices and IPI-3 devices.

The following example shows how to issue a request to enable the packet interface:

```
/*
 *    Enable the packet interface
 */

#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/cnthpi3.h>
#include <sys/pki_ctl.h>

main()
{
            int         fd;

            /*     Open the  device  */

            if  (  (fd = open("/dev/ipi3/iop_3", O_RDWR )  )  < 0  )  {
                    perror( "Unable to open the device" );
                    exit(errno);
            }

            /*   Enable the packet interface   */

            if  ( ioctl( fd, PKI_ENABLE, 0 )  <  0  )  {
                     perror( "Unable to enable the packet interface");
            }

            close(fd);
}
```

If no error has occurred, the `ioctl` return code will be 0. If an error has occurred, the error code will be one of the following codes (in decimal):

`EPKI_IOCTL_REQT`	415	`ioctl` is not valid for the device type.
`EPKI_ALREADY_ENBL`	423	The packet interface has already been enabled.

The symbols for these error codes are located in file `errno.h`.

PKI_GET_CONFIG

The PKI_GET_CONFIG request returns the IPI-3/HIPPI packet driver configuration. The configuration is returned in structure cnthpi3, which is defined in the sys/cnthpi3.h file.

The argument to the ioctl system call is a pointer to structure pki_ctl, described in the sys/pki_ctl.h file. The pki_packet field of structure pki_ctl must be set to a pointer to structure cnthpi3.

You can issue this ioctl only to the request device.

The following example shows how to issue a request for the configuration:

```
/*
 *    Get the IPI-3/HIPPI configuration.
 */

#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/cnthpi3.h>
#include <sys/pki_ctl.h>

main()
{
            struct  pki_ctl    ctl;
            struct  cnthpi3  cnt;

            int         fd;

            /*
             *     Open the request device.
             */
            if  (  ( fd = open(HPI3_REQ, O_RDWR )  )  <  0  ) {
              perror( "Unable to open the request device" );
              exit(errno);
            }

            ctl.pki_packet = (word *)&cnt;

            if  (  ioctl( fd, PKI_GET_CONFIG, &ctl )  <  0  ) {
                    perror( "Unable to get the configuration");
            }

            close(fd);
}
```

If no error has occurred, the `ioctl` return code will be 0. If an error has occurred, the error code will be one of the following codes (in decimal):

`EFAULT`	14	An address specified points outside the user's address space.
`EPKI_IOCTL_REQT`	415	`ioctl` is not valid for the device type.

The symbols for these error codes are located in file `errno.h`.

PKI_GET_DEVCONF

The `PKI_GET_DEVCONF` request returns the configuration of an IPI-3/HIPPI device. The configuration is returned in structure `cnthpi3_entry`, which is defined in the `sys/cnthpi3.h` file.

The argument to the `ioctl` system call is a pointer to structure `pki_ctl`, described in the `sys/pki_ctl.h` file. The `pki_packet` field of structure `pki_ctl` must be set to a pointer to structure `cnthpi3_entry`. The `pki_device` field of structure `pki_ctl` must be set to the name of the device.

You can issue this `ioctl` only to the request device.

The following example shows how to issue a request for the device configuration:

```
/*
 *    Get the IPI-3/HIPPI device configuration for device, d1dev
 */

#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/cnthpi3.h>
#include <sys/pki_ctl.h>

main()
{
            struct  pki_ctl              ctl;
            struct  cnthpi3_entry  ce;

            int         fd;

            /*
             *     Open the request device
             */
            if  (  ( fd = open(HPI3_REQ, O_RDWR )  )  <  0  )  {
              perror( "Unable to open the request device" );
              exit(errno);
            }

            ctl.pki_packet  = (word *)&ce
            ctl.pki_device  = 0
            strncpy( (char *)&ctl.pki_device, "d1dev", strlen("d1dev") );

            if  (  ioctl( fd, PKI_GET_DEVCONF, &ctl )  <  0 ) {
                     perror( "Unable to get the device configuration");
            }

            close(fd);
 }
```

If no error has occurred, the `ioctl` return code will be 0. If an error has occurred, the error code will be one of the following codes (in decimal).

`EFAULT`	14	An address specified points outside the user's address space.
`EPKI_IOCTL_REQT`	415	`ioctl` is not valid for the device type.
`EPKI_NO_DEVICE`	421	The device specified is not in the configuration.

The symbols for these error codes are located in file `errno.h`.

PKI_GET_DEVTBL

The `PKI_GET_DEVTBL` request returns the device table of an IPI-3/HIPPI device. The device table is returned in structure `hpi3_tab`, which is defined in the `sys/hpi3.h` file.

The argument to the `ioctl` system call is a pointer to structure `pki_ctl`, described in the `esys/pki_ctll.h` file. The `pki_packet` field of structure `pki_ctl` must be set to a pointer to structure `hpi3_tab`. The `pki_device` field of structure `pki_ctl` must be set to the name of the device.

You can issue this `ioctl` only to the request device.

The following example shows how to issue a request for the device table:

```
/*
 *    Get the IPI-3/HIPPI device table for device, d1dev
 */

#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/epack.h>
#include <sys/hpi3.h>
#include <sys/cnthpi3.h>
#include <sys/pki_ctl.h>

main()
{
            struct  pki_ctl      ctl;
            struct  hpi3_tab  tab;

            int         fd;

            /*
             *     Open the request device
             */
            if  (  ( fd = open(HPI3_REQ, O_RDWR )  )  <  0  )  {
              perror( "Unable to open the request device" );
              exit(errno);
            }

            ctl.pki_packet  = (word *)&tab
            ctl.pki_device  = 0
            strncpy( (char *)&ctl.pki_device, "d1dev", strlen("d1dev") );

            if  (  ioctl( fd, PKI_GET_DEVTBL, &ctl )  <  0 ) {
                     perror( "Unable to get the device table");
```

```
                }

                close(fd);
        }
```

If no error has occurred, the `ioctl` return code will be 0. If an error has occurred, the error code will be one of the following codes (in decimal):

`EFAULT`	14	An address specified points outside the user's address space.
`EPKI_IOCTL_REQT`	415	`ioctl` is not valid for the device type.
`EPKI_NO_DEVICE`	421	The device specified is not in the configuration.

The symbols for these error codes are located in file `errno.h`.

PKI_GET_OPTIONS

The `PKI_GET_OPTIONS` `ioctl` request returns the IPI-3/HIPPI packet driver options. The options are returned in structure `pki_option`, which is defined in the `sys/pki_ctl.h` file. Five options are returned from the driver.

Field `pki_max_async` defines the maximum number of asynchronous responses that may be enabled for an individual device. Field `pki_max_list_cmd` defines the maximum number of command list requests that may be stacked for an individual device. Field `pki_max_iop_proc` defines the maximum number of processes that can open an IOP device concurrently. Field `pki_max_non_cmdlst` defines the maximum number of noncommand list requests that may be stacked for a device. If packet driver tracing is on, field `pki_tracing` is set to 1; if it is set to 0, it is off.

To display the options, use the IPI-3/HIPPI command `hpi3_stat`(8) with option `-o`:

```
hpi3_stat -o
```

The following example shows how to issue a request for the options:

```
/*
 *    Get the IPI-3/HIPPI options
 */

#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/pki_ctl.h>
#include <sys/cnthpi3.h>

main()
{
            struct  pki_option    option;

            int         fd;

            /*
             *     Open the request device
             */
            if  (  ( fd = open(HPI3_REQ, O_RDWR )  )  <  0  )  {
              perror( "Unable to open the request device" );
              exit(errno);
            }

            if  (  ioctl( fd, PKI_GET_OPTIONS, &option )  <  0 ) {
                     perror( "Unable to get the options");
            }

            close(fd);
}
```

If no error has occurred, the `ioctl` return code will be 0. If an error has occurred, the error code will be one of the following codes (in decimal):

`EFAULT`	14	An address specified points outside the user's address space.
`EPKI_IOCTL_REQT`	415	`ioctl` is not valid for the device type.

The symbols for these error codes are located in file `errno.h`.

PKI_RECEIVE

The `PKI_RECEIVE` `ioctl` request is used to acquire the next response packet from the IPI-3/HIPPI packet driver.

The argument to the `ioctl` call is a pointer to structure `pki_ctl`, which is defined in the `sys/pki_ctl.h` file. The `pki_packet` field of structure `pki_ctl` must be set to a pointer to a buffer large enough to receive the IPI-3/HIPPI IOP response. The `pki_nbytes` field of structure `pki_ctl` must be set to the size of the buffer.

A packet will be returned only if the following conditions are met:

- The packet interface must be enabled
- A response packet must be available
- The buffer space must be large enough to accommodate the packet

You can issue this `ioctl` only to IOP devices and IPI-3 devices.

You may use the C library routine `ipi_get_pkt` to acquire packets. This routine creates and then issues a `PKI_RECEIVE ioctl` request until a packet is obtained or an error other than `EPKI_NO_PACKETS` is received. When a response packet is not available, `EPKI_NO_PACKETS` is the error code returned.

The following example shows how to receive an IOP packet:

```
/*
 *   Receive an IOP packet
 */

#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/epack.h>
#include <sys/epackk.h>
#include <sys/cnthpi3.h>
#include <sys/pki_ctl.h>

main()
{
            struct  pki_ctl    ctl;

            char pkt[HPI3_PKT_SIZE];

            int         reply;
            int         fd;

            /*
             *    Open the  IPI-3 device
             *      ipi_open is a library routine that registers for signal
             *      IPI_PKT_SIG, opens the device file, enables the packet
             *      interface, and registers signal IPI_PKT_SIG with the driver.
             */
            if  (  ( fd = ipi_open( "/dev/hpi3/d1dev" )  )  <  0  )  {
                  perror( "ipi_open" );
                  exit(-1);
             }
             ...   send a packet
             /*
              *    Poll the driver for a response packet.
              */
             bzero( pkt, HPI3_PKT_SIZE );

             ctl.pki_packet  = (word *)pkt
             ctl.pki_nbytes  = HPI3_PKT_SIZE;
             while (1)  {
                     sigoff();
                     reply = ioctl( fd, PKI_RECEIVE, &ctl );
                     if  ( !reply ||  (errno != EPKI_NO_PACKETS)  )
                              break;
```

```
                pause();
        }

        sigon();
           .
           .
        close(fd);
}
```

If no error has occurred, the `ioctl` return code will be 0. If an error has occurred, the error code will be one of the following codes (in decimal):

`EFAULT`	14	An address specified points outside the user's address space.
`EPKI_TOO_LARGE`	403	The response buffer is not large enough to accommodate the response packet.
`EPKI_NOT_ENABLED`	406	The packet interface has not been enabled.
`EPKI_IOCTL_REQT`	415	`ioctl` is not valid for the device type.
`EPKI_RESPBUF_LOST`	429	That part of the packet returned in a response buffer was lost.

The symbols for these error codes are located in file `errno.h`.

PKI_SEND

The `PKI_SEND` request is used to send request packets to the IPI-3/HIPPI packet driver.

The argument to the `ioctl` system call is a pointer to structure `pki_ctl`, described in the `sys/pki_ctl.h` file. The `pki_packet` field of structure `pki_ctl` must be set to a pointer to a valid IPI-3/HIPPI IOP request. The `pki_nbytes` field of structure `pki_ctl` must be set to the size of the IPI-3/HIPPI request packet. The size of the packet cannot exceed `EPAK_MAXLEN`.

The packet will be accepted only if the following conditions are met:

- The packet interface must be enabled
- The request cannot exceed the request limit
- The request code must be a valid EIOP request code
- The request must be valid for the device
- The IOP driver must already be started

The request packet must be of a format defined in the `sys/epackk.h` file.

You can issue this `ioctl` request only to IOP devices and IPI-3 devices.

You may use the C library routine `ipi_put_pkt` to send packets.

The following example shows how to send an IOP packet:

```
/*
 *   Send an iop packet
 */

#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/epack.h>
#include <sys/epackk.h>
#include <sys/cnthpi3.h>
#include <sys/pki_ctl.h>

main()
{
            epackk  *pk;
            struct  pki_ctl    ctl;

            char  pkt[HPI3_PKT_SIZE];

            int         fd;

            /*
             *   Open the  IPI-3 device
             *     ipi_open is a library routine that registers for signal
             *     IPI_PKT_SIG, opens the device file, enables the packet
             *     interface, and registers signal IPI_PKT_SIG with the driver.
             */
            if  (  ( fd = ipi_open("/dev/hpi3/d1dev" )  )  <  0  )  {
                  perror( "ipi_open" );
                  exit(-1);
            }

            /*
             *   Create the IOP request
             */
            bzero( pkt, HPI3_PKT_SIZE );

            pk = (epackk *)pkt;
            pk->ek_open_stream.ek_request = EK_OPEN_STREAM;

            /*
             *    Send the request.
             */
            ctl.pki_packet  = (word *)pk
            ctl.pki_nbytes  = sizeof(ek_open_stream);

            if  (  ioctl( fd, PKI_SEND, &ctl )  <  0 ) {
                     perror( "Unable to send the packet");
            }

            close(fd);
}
```

If no error has occurred, the `ioctl` return code will be 0. If an error has occurred, the error code will be one of the following codes (in decimal):

`EFAULT`	14	An address specified points outside the user's address space.
`EINVAL`	22	The packet length is not in the range 1 through `EPAK_MAXLEN`. Or, on a request that requires a data transfer, the memory type is not valid.
`EPKI_ASYNCH_LIM`	404	Cannot enable more asynchronous responses than the maximum allowed.
`EPKI_INVAL_CODE`	405	Request code specified is not valid.
`EPKI_NOT_ENABLED`	406	Packet interface is not enabled.
`EPKI_REQ_LIM`	407	Exceeded the maximum number of requests allowed.
`EPKI_BAD_RESYNC`	408	A command list request was issued with a resynchronize code that does not match the resynchronize code of the last command list response.
`EPKI_NO_START`	409	The IOP driver has not been started.
`EPKI_REQT_TYPE`	414	Request code specified is not valid for the device type.
`EPKI_IOCTL_REQT`	415	`ioctl` is not valid for the device type.
`EPKI_IOP_SEND`	416	The packet driver could not send the packet to the IOP.
`EPKI_DEVS_ACTIVE`	418	Cannot stop an IOP driver that has an active device.
`EPKI_CMDLIST_LIM`	430	Exceeded the maximum number of command list requests allowed.

The symbols for these error codes are located in file `errno.h`.

PKI_SIGNO

The `PKI_SIGNO` request is used to register for a signal that will be sent to the user when the packet driver receives a response from the IOP.

The argument to the `ioctl`(2) system call is a pointer to structure `pki_ctl`, described in the `sys/pki_ctll.h` file. The `pki_signo` field of struct `pki_ctl` must be set to the signal number.

Before registering for a signal (PKI_ENABLE), you must enable packets.

You can issue this request only to IOP devices or IPI-3 devices.

The following example shows how to issue a request to register a signal by using the packet driver.

```
/*
 *    Register a signal by using the packet driver.
 */

#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/signal.h>
#include <errno.h>
#include <sys/pki_ctl.h>

main()
{
            struct  pki_ctl      ctl;

            int         fd;

            /*   Register for the packet-available signal   */

            if ( sigctl( SCTL_REG, IPI_PKT_SIG, 0 ) < 0 ) {
                   exit(errno);
            }

            /*     Open the  device  */

            if ( (fd = open("/dev/ipi3/d1dev", O_RDWR ) < 0 ) {
                    perror( "Unable to open the device" );
                    exit(errno);
            }

            . . . enable packets (see PKI_ENABLE)

            ctl.pki_psigno  =  IPI_PKT_SIG;
            if  (  ioctl( fd, PKI_SIGNO, &ctl )   <  0  )  {
                     perror( "Unable to register for a signal");
            }

            close(fd);
}
```

If no error has occurred, the `ioctl` return code will be 0. If an error has occurred, the error code will be one of the following codes (in decimal):

`EFAULT`	14	The `ioctl` argument specified points outside the user's address space.
`EINVAL`	22	The signal number specified is not within the range 0 to NSIG.
`EPKI_NOT_ENABLED`	406	The packet interface has not been enabled.
`EPKI_IOCTL_REQT`	415	`ioctl` is not valid for the device type.

The symbols for these error codes are located in file `errno.h`.

SEE ALSO

`hpi3_clear`(8), `hpi3_config`(8), `hpi3_option`(8), `hpi3_start`(8), `hpi3_stat`(8), `hpi3_stop`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`hpm` – Hardware Performance Monitor interface

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

Cray PVP systems contains a Hardware Performance Monitor (HPM) that counts certain activities within a CPU when it is executing in user mode. The HPM driver interfaces let a user read the performance counters for the groups of processes or change to a different monitor group number. No group number change is supported on the CRAY C90 and CRAY T90 series.

The HPM driver supports the following three minor devices:

`/dev/hpm`	Minor device 0; returns data about the current process.
`/dev/hpm_mult`	Minor device 1; returns data about the current process and any other processes that are or have been in a multitasking group with the current process.
`/dev/hpm_all`	Minor device 2; returns data about all processes (systemwide) that are running or have been disconnected.

The HPM driver supports the `ioctl`(2), `open`(2), and `read`(2) system calls. The `read`(2) system call returns only `sizeof(struct hpm)` bytes on minor devices 0 and 1, and `(sizeof(struct hpm)*NCPU)` bytes on minor device 2. If the requested number of bytes is fewer than this, an error is returned. If more than this is requested, the request is truncated.

On Cray PVP systems (except CRAY C90 and CRAY T90 series), the format of the `ioctl` request is as follows:

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

ioctl(fildes,HPMSET,group)
```

The only valid `ioctl` requests are `HPMSET` on minor device 0 and 2. Super-user permission is required to perform an `HPMSET` on minor device 2. `HPMSET` accepts as an argument the HPM group selected. Valid HPM groups are 0, 1, 2, or 3. `HPMSET` on minor device 2 changes the value of the global HPM group to the group specified. All processes that have not explicitly chosen an HPM group are put into the global group the next time they are connected. Group 1 is the default for the global group.

The possible errors for the system calls are as follows:

`EFAULT`	Returned if the read buffer is not entirely within the user field.
`EINVAL`	Returned if the selected group is outside the allowable range, if the read buffer is too small, or if an `ioctl` call other than `HPMSET` is issued.

`ENXIO`	Returned if requested on other than the CRAY J90 series or if a minor device other than 0, 1, or 2 is requested on an `open`(2), `close`(2), or `read`(2), or if a minor device other than 0 or 2 is requested on an `ioctl`.
`EPERM`	Returned if an `HPMSET` on minor device 2 is tried and the user does not have super-user permissions.

NOTES

The system always maintains the counters on any mainframe that supports the HPM. Except for the CRAY C90 and CRAY T90 series, a user program usually starts in group 1, although the group number is inherited from the parent and can be overridden by setting the global group.

On the Cray PVP systems (except for the CRAY C90 and CRAY T90 series), accounting of execution time for autotasked and microtasked programs does not charge for time spent waiting on a semaphore. The group number defaults to group 1 to gain this information. If an autotasked or microtasked program runs in a different HPM group, the system does not have the information necessary to avoid charging for wait-semaphore time. Therefore, running an autotasked or microtasked program with a group other than 1 shows nonrepeatable execution times for the program. On the CRAY C90 and CRAY T90 series, wait-semaphore time is always included.

The definition of what is counted differs between the CRAY C90 and CRAY T90 series and the CRAY J90 series. For more information, see the functional description manual for your mainframe.

FILES

`/dev/hpm`

`/dev/hpm_all`

`/dev/hpm_mult`

`/dev/MAKE.DEV`

`/usr/include/sys/hpm.h`

SEE ALSO

`hpm`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`ioctl`(2), `open`(2), `read`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`hpmall`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

Guide to Parallel Vector Applications, Cray Research publication SG–2182

NAME

`hsx` – High-speed External Communications Channel interface

IMPLEMENTATION

All Cray Research systems except CRAY J90 series and CRAY EL series

DESCRIPTION

The `hsx` interface (or `hsx` driver) drives a High-speed External (HSX) Communications Channel. Application processes use the `hsx` driver by means of the standard UNICOS `close`(2), `ioctl`(2), `listio`(2), `open`(2), `read`(2), `reada`(2), `write`(2), and `writea`(2) system calls. Each of the special files in the `/dev/hsx` directory represents one input or output HSX channel.

By convention, HSX file names have the following format:

```
/dev/hsxn/ixx
/dev/hsxn/oxx
```

n	Physical channel number
`i`	Input channel
`o`	Output channel
xx	Logical channel number

For operation in one direction, only one channel must be opened. If the application uses both read and write operations, it must open both an input and an output channel. A read operation is valid only on an input channel; a write operation is valid only on an output channel. All HSX I/O is raw; the user process is locked in memory as data moves directly between the user buffer and the channel. User buffers must therefore be word-aligned, and their length must be a multiple of 8 bytes.

Dedicated and Shared HSX Channels

The `hsx` driver supports two modes of operation: dedicated and shared. The mode is assigned to each channel when it is configured. If a channel is dedicated, only one process can have that channel open at a time. If a channel is shared, several processes can use one HSX channel. The driver enforces a protocol on the shared channel; this allows it to determine the destination of each incoming message.

CRAY Y-MP systems that are configured to read and write the HSX channels with SSD solid-state storage buffers do not support shared channels. For more information on configuring HSX channels on Cray PVP systems, see *UNICOS Networking Facilities Administrator's Guide*, Cray Research publication SG–2304.

HSX `ioctl` Requests

Several `ioctl` requests are available. They have the following format:

```
#include <sys/hx.h>
ioctl (fildes, command, arg)
struct hxio *arg;
```

The valid `ioctl` requests are as follows:

`HXC_CLR`	Sends a clear signal on the output channel. This request is restricted to dedicated channels.
`HXC_EXC`	Sends an exception signal on the input channel. This request is restricted to dedicated channels.
`HXC_GET`	Gets the current driver parameter settings; stores them in the `hxio` structure referenced by *arg*.
`HXC_SET`	Sets driver parameters from the structure referenced by *arg*.
`HXC_WFI`	Waits for interrupt: blocks until a clear signal (from an input channel) or exception signal (from an output channel) is received. If the channel signal has already been received, the request returns immediately. This request is restricted to dedicated channels.

The `hxio` structure contains the following fields:

`unsigned int flags` Flags that control channel operation. The *flags* field controls the driver's options. The bits in the flags field are defined as follows:

`HXCF_ACM`	Sets the alternative checkbyte mode (for an output channel) or disables SECDED (for an input channel).
`HXCF_DBG`	Sets debug mode. On write operations, the data is discarded. On read operations, the input data is undefined. In debug mode, the driver does everything except actual I/O on the HSX channel. HSX channel hardware does not have to be present for this mode to be used.
`HXCF_DED`	The channel is open in dedicated mode; no other processes may share the channel. If this flag is set, the value in the *path* field is meaningless. The `ioctl` request `HXC_GET` is used to set this flag. The `ioctl` request `HXC_SET` ignores this flag.
`HXCF_DOWN`	If this bit is set, the channel is unavailable to other users. If this bit is clear, the channel is available. When the channel is down, the driver returns the error `ENXIO`. Programs that reconfigure the channel should issue a `close` request immediately after the `ioctl` request, then reopen the channel before trying to do anything else. This bit is set in the `ioctl` request `HXC_SET` on a dedicated channel. `HXC_SET` is a read-only flag.

`int err`	Detailed error status from the last request; for a list of HSX errors, see the MESSAGES section.
`int path`	Logical path number for device (ignored on `HSC_SET`).
`unsigned int tmo`	Time-out value (in seconds).

listio Special Features

The hsx driver defines two flags to be used in the li_drvr field of the listreq structure (see listio(2)):

HXLI_CHD	Indicates, in effect, that user data is chained. When this flag is set, the driver suppresses the end-of-block signal at the end of the current list entry (for an output channel) or allows the current input data block to overflow into the next list entry (for an input channel).
HXLI_STRB	Specifies that each stride (in memory) is a block. (A *stride* is the distance from the start of one section of data to the start of the next section.) This flag is valid only on dedicated channels and is meaningful only when the number of strides in a list entry is greater than 1. When the HXLI_STRB flag is set, HXLI_CHD is ignored.

If the driver encounters an end-of-block signal during a read operation in which the HXLI_CHD flag is set, no error indication is returned to the user. Subsequent read operations will complete with a data length of 0 until the HXLI_CHD flag is cleared. If the user issues the ioctl request HXC_GET to check the status after a chained read operation, the err field of the hxio structure is set to the error code HXST_EOB, indicating that an end-of-block signal arrived before the last read request was processed.

If the driver has not detected an end-of-block signal on a unchained read operation (that is, read(2), reada(2), or listio(2) with the HXLI_CHD flag clear) by the time the read operation completes, the driver discards the unread portion of the block. The byte count returned to the user is equal to the size of the buffer, and no error code is returned in errno. If the user issues the ioctl request HXC_GET to check the status after a unchained read operation, the err field of the hxio structure is set to the error code HXST_LONG, indicating that the entire block was not read and the remainder was discarded.

When using the listio chaining feature on shared channels to combine more than one buffer to make one data block, the user must include all buffers in the same listio(2) system call. That is, ensure that the HXLI_CHD flag is clear in the li_drvr field of the last item for an HSX channel in each listio(2) call. Failure to do this can cause lost or corrupt data on the channel if the user process is swapped between requests.

HSX Protocol

The hsx driver enforces no protocol on dedicated channels. On shared channels, the driver assigns a logical path value to each device configured on the shared channel. The ioctl request HXC_GET returns the logical path value for a shared channel in the *path* field of the hxio structure.

Each block of data must begin with a word that contains the logical path values of the sending and receiving channels. The first word must have the following format (as defined in the sys/hx.h include file):

```
struct hxhdr {
        unsigned int    unused  :32;    /* not used by driver   */
        unsigned int    to      :16;    /* destination address  */
        unsigned int    from    :16;    /* source address       */
};
```

The driver uses only two fields in the word. These fields have the following meanings:

`to`	Identifies the channel to receive the message. The driver looks at this field in each incoming block and delivers the block to the process reading the channel assigned to the number in this field.
`from`	Identifies the channel sending the message. In a message on an output channel, this field contains the value assigned at configuration time to the special file in `/dev` that represents the device.

You can use any protocol that fits this template on shared HSX channels. The sending process must set the `to` field in the `hxhdr` structure to the correct destination address. If no process is reading the destination device for incoming data, the driver discards the block after a period of time. During this time interval, no data can flow on the HSX channel.

Software Loopback Feature

The `hsx` driver contains a software loopback feature to debug application codes. The software loopback feature works on the `/dev` HSX files that are configured as HSX software loopback channels.

Loopback channels come in pairs; an even-numbered channel (*n*) is paired with an odd-numbered channel (*n*+1). Data written on the odd channel can be read on the even channel. A pair of channels simulates one set of HSX channels cabled in loopback configuration.

MESSAGES

The `hsx` driver returns one of the following error codes in `errno` on a fatal error. The error codes and their meanings are as follows:

`EFAULT` An `ioctl` request specified a bad argument address.

`EINVAL` The driver software has detected a fatal parameter error. Errors in system configuration and user errors in system call invocation can cause this error.

`EIO` One of the following conditions can cause this error:

- A fatal I/O error occurred or the HSX channel closed while asynchronous I/O was active (on a `read`(2) or `write`(2) system call).
- A data block was too long for the input buffer (on a `read`(2) system call) or overflowed the channel (on a `write`(2) system call).
- An I/O request timed out (on a `read`(2) or `write`(2) system call).

The detailed error status is available in the `err` field of the `hxio` structure; to read this field, use the `ioctl` request `HXC_GET`.

`ELATE` An input request timed out.

`ENXIO` The HSX channel is unavailable (on an `open`(2) system call), or the channel is not open (on a `close`(2) system call). This code can mean that the IOP did not allocate some resource.

After any fatal error, the detailed error status is retrieved by using the `ioctl` request `HXC_GET`. The driver returns error codes in the `err` field of the `hxio` structure.

`HXST_ABRT`	Channel abort (output only).
`HXST_BUF`	IOS I/O buffer unavailable on open operation.
`HXST_CHAN`	CPU gave bad channel number to IOS; caused by configuration error.
`HXST_CLR`	Clear pulse received (input channel only).
`HXST_DATA`	SECDED error or lost data (input only).
`HXST_DBG`	Debug mode error; indicates a driver fault (should never occur).
`HXST_EOB`	Unexpected end-of-block (input only).
`HXST_FLGS`	Buffer flags do not match; indicates a driver fault (should never occur).
`HXST_FMEM`	IOS free memory unavailable on open operation.
`HXST_FNC`	Illegal function code; indicates a driver fault (should never occur).
`HXST_HISP`	No high-speed channel to this IOP; caused by configuration error or wrong target memory.
`HXST_LLEN`	Transfer length too long; indicates a driver fault (should never occur).
`HXST_LONG`	Long block received (input only).
`HXST_MOS`	MOS buffer unavailable on open operation (debug mode only).
`HXST_NDEV`	No device present on the channel (hardware signal).
`HXST_OK`	No error (binary 0).
`HXST_OPEN`	Channel is not open; indicates a driver fault (should never occur).
`HXST_OVER`	Data-overrun error (input only).
`HXST_TM`	Bad target memory type; indicates a driver fault (should never occur).
`HXST_TMO`	Request timed out.
`HXST_XDT`	Exception pulse received during transfer (output only).
`HXST_ZLEN`	Buffer length is 0; indicates a driver fault (should never occur).

FILES

`/dev/hsx`*n*`/*`	HSX channel special files
`/usr/include/sys/hpacket.h`	
`/usr/include/sys/hx.h`	
`/usr/include/sys/hxsys.h`	

SEE ALSO

`close`(2), `ioctl`(2), `listio`(2), `read`(2), `reada`(2), `write`(2), `writea`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`hsxconfig`(8), `mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`hy` – HYPERchannel adapter interface

IMPLEMENTATION

Cray PVP systems except CRAY J90 series

DESCRIPTION

The HYPERchannel driver provides an interface for NSC HYPERchannel adapters connected to the IOS. The HYPERchannel driver also is used for Cray Research front-end interfaces (FEIs) and VME interfaces attached to the IOS. For more information, see `fei`(4) and `vme`(4). The driver accepts standard UNICOS `close`(2), `listio`(2), `open`(2), `read`(2), `reada`(2), `write`(2), and `writea`(2) system calls; it also accepts `ioctl`(2) requests.

By convention, the HYPERchannel special file names are in the following format:

`/dev/comm/`*address*`/lp`*nn*

address	The address is composed of three elements: the interface type, the IOS number, and the channel number (in octal). The interface type is indicated by one character: `f` (FEI), `n` (NSC), or `v` (VME). The IOS number is either 0 or 1. The channel number is the octal channel to which the interface is connected on the specified IOS.
nn	Logical path number for the device (0 through 15).

For each device, up to 16 logical paths are available. The minor number is the logical path across a device. For example, if the site has configured 3 network interfaces that use the `hy` driver with 16 logical paths for each, the first device in the `comm_info` of type `COMM_HYDRIVER` uses logical paths 0 through 15. The second device uses logical paths 16 through 31. The third device uses 32 through 47.

You cannot open a minor device when it is already open. Any attempt to do so fails with an `EBUSY` error.

You can use the `ioctl` requests `HYGET` and `HYSET` to change parameters for the IOS. Kernel-level routines can use the `HYKGET` and `HYKSET ioctl` requests to change parameters for the IOS. To return various kinds of driver and HYPERchannel status, use the `HYSTAT` function. The subfunctions are defined in the `npstat` structure in `sys/hy.h`. The last status is the IOS driver's status and is specific to that driver. The `HYHLTIO ioctl` request allows users to halt outstanding driver packet I/O requests. Because this request must be issued by the same process that issued the I/O, you can use it only with asynchronous I/O.

The ioctl structure is defined in the sys/hy.h include file, as follows:

```
    struct npstat {
            int     mad;    /* maximum associated data size         */
            int     wtmo;   /* write timeout                        */
            int     rtmo;   /* read timeout                         */
            int     inq;    /* input messages to queue              */
            int     path;   /* adapter path requested               */
            int     lrt;    /* last response time                   */
            int     nperr;  /* last N packet error return from IOS  */
                            /* (valid only when the last operation  */
                            /* returned an error)                   */
            int     sfunc;  /* status subfunction see npstats.h     */
            char    *sbuf;  /* status buffer pointer                */
    } ;
```

`mad`	Maximum associated data size (in words); this parameter places a limit on the message size that the IOS expects.
`wtmo`	Write time-out.
`rtmo`	Read time-out (in tenths of a second) for both read and write operations. The IOS queues up to four messages before discarding input from the adapter that the mainframe has not read.
`inq`	Input messages to queue; allows a process to specify up to four messages to queue. A process can request a specific minor device number. This is currently unused.
`path`	Adapter path.
`lrt`	Last response time; the time since the last packet was received from the IOS for this minor device.
`nperr`	Last error status from the IOS; valid only when the last I/O operation returns an error. For a list of error response codes from the IOS and their meaning, see the `nperr` Values subsection.
`sfunc`	The statistics subfunction to be used; used only in conjunction with the `HYSTAT ioctl` request.
`sbuf`	The address of the driver statistics buffer; used only in conjunction with the `HYSTAT ioctl` request.

You can use an `ioctl` request after the `open` operation and before the first `read` or `write` operation to change the IOS parameters.

The `read`(2) and `write`(2) system calls issued by a process to the HYPERchannel driver differ from typical `read`(2) and `write`(2) system calls only in that the first 64 bytes of the data must be a HYPERchannel message proper. The message proper is defined as follows:

```
struct mp {
        char    control[2];     /* NSC control word           */
        char    acode[2];       /* NSC access code            */
        char    to[2];          /* NSC destination adapter    */
        char    from[2];        /* NSC source adapter         */
        char    param[56];      /* NSC parameters             */
} ;
```

The details of the contents of message proper fields are available in NSC documentation.

If a `read`(2) system call is not satisfied within the time-out period, an `ELATE` error is returned. If a `write`(2) system call cannot be completed, it is retried periodically in the time allowed by the time-out period before an `ELATE` error is returned. A `close`(2) system call closes the minor device. A `CEIO` error terminates any outstanding system calls.

`nperr` Values

The following tables list the possible values of `nperr`, according to IOS driver (a response of 0 always means "no error").

The following octal status values have the associated meanings for all N-packet drivers:

Value	**Definition**
03	Protocol error concerning N-packet request order.
04	Bad channel, owner, or path.
05	Bad function code.
11	Cannot create activity; channel, path, or memory not available.
12	Error on configuration request processing.
54	Path terminated.

The following octal status values have the associated meanings only for FEI drivers:

Value	**Definition**
40	Message length is 0 or too big.
41	Read time-out.
43	Write time-out.
44	Write sequence error.
50	Illegal function or subfunction.
52	MOS or local memory not available; cannot create.

53	Port select error.
54	Driver terminating.
56	Message read is too short; data read is too big.
57	Insufficient space allocated in CPU for input.
60	Parity error received.
61	Read sequence error.

The following octal status values have the associated meanings only for VME and NSC drivers:

Value	**Definition**
36	Path 0 not available for loopback destination.
41	Read time-out.
43	Write time-out.
45	No read for loopback write.
51	No remote adapter ID in write message.
52	MOS or local memory not available.
57	Insufficient space allocated in CPU for input.
50	Illegal function or subfunction.
56	Transfer length specified on write is not valid.
60	Loopback write error.
62	Bad CPU address given in N-packet.

The following octal status values have the associated meanings only for VME drivers:

Value	**Definition**
42	Input channel time-out.
44	Output sequence error.
52	Cannot create activity.
53	Output channel time-out.
57	Parity error, message bad size, or data not present but expected.

The following octal status values have the associated meanings only for NSC drivers:

Value	**Definition**
42	Read error.
44	Associated data flag is set, but no associated data is available.
53	Remote adapter not available; write aborted.

61	Residual data on channel after data read.
63	Input sequence error.
64	Input parity error.

The following octal status values have the associated meanings for COMM drivers (12-Mbyte interconnect specification drivers):

Value	**Definition**
20	CPU I/O request time-out.
21	Driver detected function code error.
22	Driver termination in progress.
23	Insufficient space allocated by CPU.
30	Parameter block detected is not valid.
31	Data detected is not valid.
32	Write time-out.
33	Driver detected parity or sequence error.
34	I/O channel time-out.
35	Overrun (input channel).
36	Transfer length error.

BUGS

The NSC status is not available from the IOS. Currently, no special functions of the adapter are supported.

FILES

`/dev/comm/*/*`

`/usr/include/sys/hy.h`

`/usr/include/sys/hysys.h`

SEE ALSO

`fei`(4), `vme`(4)

`ioctl`(2), `listio`(2), `read`(2), `reada`(2), `write`(2), `writea`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`inode` – inode file system

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `/inode` file system allows privileged processes access to a file or directory when the process knows the device and inode number of a file system. The `/inode` file system has no files or storage but if it is asked to translate a path of the form `/inode/`*ddd.iii.ggg.ff* (where *ddd* is the device number, *iii* is the inode number, *ggg* is the optional generation number, and *ff* is an optional list of flags `r` or `w`), it returns the vnode for the specified file or directory. If the generation number is provided it must match the current generation number.

Access to the */inode* path translation is limited to privileged processes: Root user for systems with traditional UNIX security or users with an active `secadm` category for systems using privilege assignment lists (PALs).

Executing an `ls`(1) or `readdir`() on the `/inode` file system root directory shows only the "." and ".." directories.

Since `/inode` has no storage or files but serves only as a path to other file systems, once the path element following the `/inode` is translated, the normal rules for file system access for that target item apply.

CONFIGURATION

You create an `/inode` file system with the following steps:

1. Create an empty directory called `/inode` with the following command:

   ```
   mkdir /inode
   ```

2. Modify the system mount scripts and `/etc/fstab` file so that the `/inode` file system is always mounted at system startup.

   `fstab` entry:

   ```
   /inode /inode INODE
   ```

NOTES

The use and implementation of the `ioctl`(2) operations documented in this entry are subject to change in future releases of UNICOS.

FILES

`/inode`	`/inode` file system root

SEE ALSO

`fstab`(5)

`mount`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

iosE – IOS model E interface

IMPLEMENTATION

Cray PVP systems with an IOS model E

DESCRIPTION

The /dev/iosE/* device is used as the interface to the IOS model E (IOS-E). The ioctl(2) system call issues requests to the ioscntl device driver.

The minor number of a device specifies a particular IOP (for example, the /dev/iosE/iop.1.3 device has a minor number of 11, which indicates cluster 1, eiop 3).

The different structures used for the ioctl(2) system call are defined in the /usr/include/sys/ioscntl.h include file, and they are described as follows:

```
#define get_cluster(x)  (int)(((uint)x>>3)&0377)  /* get cluster from minor */
#define get_eiop(x)     (int)(x&07)               /* get eiop from minor    */

/*
 *     Ioctl argument structure
 */
struct ioscntl {
        union {
        int i;
                char *cp;
                word *wp;
                struct ioscntl_cf *cf;
        } ios_arg1;
        union {
                int i;
                word *wp;
        } ios_arg2;
        int ios_arg3;
        int ios_arg4;
};
/* Field equivalence defines */
#define      ios_hisp_no     ios_arg1.i
#define      ios_channel     ios_arg2.i
#define      ios_mode        ios_arg3
#define      ios_target      ios_arg4
#define      ios_path        ios_arg1.cp
#define      ios_send        ios_arg1.wp
#define      ios_receive     ios_arg2.wp
#define      ios_length      ios_arg3
#define      ios_iop_config  ios_arg1.cf
/*
```

```
 *     IO_GET_CONFIG buffer structure.
 */
struct ioscntl_cf {
        int ios_status;                 /* configuration status bits         */
        char ios_bootpath[IO_PATHMAX];  /* partial path name for boot binary */
        int ios_usage;                  /* drivers currently using IOP       */
        /* the following fields are valid only for the MUXIOP                */
        int ios_lowsp;                  /* low-speed channel number          */
        struct {
                int open;               /* open flag                         */
                int channel;            /* channel number                    */
                int target;             /* the target memory for the channel */
                int mode;               /* operating mode for the channel    */
        } ios_hisp[2];
};
```

Configuration requests affect future I/O in the following ways:

`IO_STOP`	New I/O to the target IOP is queued but not processed. Current I/O is not guaranteed to complete.
`IO_ABORT`	All current and new I/O to the target IOP is rejected.
`IO_RESTART`	Any current I/O that has not completed is requeued. All queued I/O to the target IOP is then processed.

The following is a description of available `ioctl` requests:

`IO_SET_PATH` Specifies the path name for a binary file that will be booted later into the target IOP by using the `IO_BOOT_IOP` request. This request can be done only to a IOP that has not been booted.

Required argument:

ios_path	Pointer to the path name of the boot binary file

`IO_SET_LOWSP` Sets the low-speed channel number. This request can be done only when the low-speed channel is down.

Required argument:

ios_channel	Low-speed channel number

`IO_SET_HISP` Sets the HISP channel parameters. This request can be done only when the MUXIOP for the specified cluster is up and accessible through the low-speed channel, and the HISP is down.

Required arguments:

ios_hisp	HISP ordinal; 0 for HISP0, 1 for HISP, and so on.
ios_channel	HISP channel number.

ios_mode	HISP channel mode.
ios_target	HISP channel target memory.

`IO_BOOT_IOP` Boots the specified IOP. The binary file used to boot the IOP must have been specified by a previous `IO_SET_PATH` request.

`IO_DOWN_IOP` Sets the specified IOP to a down state. This causes the `IO_ABORT` condition to be set for the target IOP.

`IO_RDOWN_IOP` Sets the specified IOP to a down state in a restartable way. This causes the `IO_STOP` condition to be set for the target IOP.

`IO_UP_LOWSP` Sets the low-speed channel to an up state. This request can be done only when the low-speed channel is down. This causes the `IO_RESTART` condition to be set for those IOPs in the cluster that were stopped by a previous `IO_DOWN_LOWSP` request.

`IO_DOWN_LOWSP` Sets the low-speed channel to a down state. This request can be done only when the low-speed channel is up. This causes the `IO_STOP` condition to be set for those IOPs in the cluster that are processing requests.

`IO_UP_HISP` Sets a HISP channel to an up state. This request can be done only when the HISP channel is down.

Required argument:

ios_hisp	HISP ordinal; 0 for HISP0, 1 for HISP, and so on.

`IO_DOWN_HISP` Sets a HISP channel to a down state. This request can be done only when the HISP channel is up.

Required argument:

ios_hisp	HISP ordinal; 0 for HISP0, 1 for HISP, and so on.

`IO_UP_INTER` Sets an inter-IOP channel to an up state. This causes the `IO_RESTART` condition to be set for IOP if it was processing requests at the time of a previous `IO_DOWN_INTER` request.

`IO_DOWN_INTER` Sets an inter-IOP channel to a down state. This causes the `IO_STOP` condition to be set for IOP if it is processing requests.

`IO_ECHO` Echoes message through an IOP.

Required arguments:

ios_send	Word pointer to the send buffer
ios_receive	Word pointer to the receive buffer
ios_length	Number of words in the echo message

`IO_GET_CONFIG` Gets an IOP configuration.

Required argument:

ios_iop_config	Pointer to the buffer structure to receive the data

In addition to the standard `ioctl` error codes (see `ioctl`(2)), the following errors cause an `ioctl` request to fail:

`[EFAULT]`	Bad address passed to system call.
`[EAGAIN]`	Attempt to down an inter-IOP channel that is already down.
`[ENXIO]`	MUXIOP or IOP is not in a state that allows configuration of an inter-IOP channel.
`[EBUSY]`	MUXIOP or IOP is currently processing another `ioctl`(2) request.

NOTES

Only the super user can use the `/dev/ios` interface.

FILES

```
/dev/ios
/usr/include/sys/epack.h
/usr/include/sys/ioscntl.h
```

SEE ALSO

`ioctl`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

ipi3 – IPI-3/IPI interface

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The IPI-3 interface is used as a connection to the IPI-3/IPI IOPs and the IPI-3/IPI devices. These devices are represented by special files in the /dev/ipi3 directory.

The ASCII names for the IOP devices and the IPI-3/IPI devices in the /etc/config/ipi3_config configuration file are used to create the special files in /dev/ipi3.

The pki_ctl structure, as defined in the sys/pki_ctl.h include file, is used to communicate between the packet driver and the controlling process. The packet driver control structure is defined as follows:

```
struct pki_ctl{
       int    pki_psigno;                /* Signal to receive        */
       word   *pki_packet;               /* Packet from user program */
       int    pki_nbytes;                /* Length of packet         */
       int    pki_device;                /* Device name              */
       }
```

The following is a list of the ioctl(2) requests used with the IPI-3/IPI interface:

PKI_CLEAR	Clears the IPI-3/IPI device.
PKI_DRIVER_STS	Returns the status of the IPI-3/IPI packet driver.
PKI_ENABLE	Enables a packet interface.
PKI_GET_CONFIG	Returns the IPI-3/IPI configuration.
PKI_GET_DEVCONF	Returns the configuration of a device.
PKI_GET_DEVTBL	Returns a driver device table.
PKI_GET_OPTIONS	Returns the options of the IPI-3/IPI packet driver.
PKI_RECEIVE	Returns an IPI-3/IPI packet.
PKI_SEND	Sends an IPI-3/IPI packet.
PKI_SET_OPTIONS	Modifies the options of the IPI-3/IPI packet driver.
PKI_SIGNO	Registers the signal to be sent to the user when an interrupt is received from the IOP.

FILES

`/dev/ipi3/reqt`	IPI-3/IPI interface device
`/usr/include/sys/pki_ctl.h`	Structure definition of `pki_ctl`

SEE ALSO

`reqt`(4)

`ipi3_clear`(8), `ipi3_config`(8), `ipi3_option`(8), `ipi3_start`(8), `ipi3_stat`(8), `ipi3_stop`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

Tape Subsystem Administration, Cray Research publication SG–2307

NAME

`ldd` – Logical disk device

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The files in `/dev/ldd` are logical disk descriptor files (see `ldesc`(5)). They are used to combine other logical or physical disk slices into a single logical disk device. Typically, the files in `/dev/ldd` are referenced by name by a disk block special file in `/dev/dsk` that has a major device number of the concatenated logical disk driver, ldd. See `dsk`(4). Concatenated logical disk devices are differentiated from other logical disk devices by the major device number defined in `/usr/src/uts/c1/cf/devsw.c`.

Usually, a logical descriptor file is used to combine physical disk slices in the following manner (see `pdd`(4)):

```
mknod /dev/ldd/usr L /dev/pdd/usr.0 /dev/pdd/usr.1
```

When the `/dev/ldd/usr` file is referenced by a character or block special device, its member slices, `/dev/pdd/usr.0` and `/dev/pdd/usr.1`, are combined by a logical disk driver into a single logical disk device.

Typically, a block special file in `/dev/dsk` references a file in `/dev/ldd`. Following through with the preceding example, you can make a simple concatenated logical device by using the `mknod`(8) command (see `mknod`(8) and `dsk`(4)). In this example, the major device number is `dev_ldd`, and the minor device number is 12. The two 0’s are placeholders and are unused.

```
mknod /dev/dsk/usr b dev_ldd 12 0 0 /dev/ldd/usr
```

The following logical disk devices are supported:

`dev_ldd`	The simple concatenated logical device. The physical slices are concatenated to form a single logical disk device. See `dsk`(4).
`dev_mdd`	A mirrored logical disk device. Two or more disk slices are identical copies of one another for data redundancy. See `mdd`(4).
`dev_sdd`	A striped logical disk device. Two or more disk slices are striped to increase bandwidth. See `sdd`(4).

The logical disk devices are differentiated by their major device numbers defined symbolically in `/usr/src/uts/c1/cf/devsw.c`.

FILES

`/dev/dsk/*`

`/dev/ldd/*`

`/usr/include/sys/ldesc.h`

SEE ALSO

`dsk`(4), `ldesc`(5), `mdd`(4), `pdd`(4), `sdd`(4), `ssdd`(4)

`ddstat`(8), `mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`lo` – Software loopback network interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The loop interface, `lo`, is a TCP/IP pseudo device. It is a software loopback mechanism, which you can use for performance analysis, software testing, and/or local communication. By default, the loopback interface is accessible at address 127.0.0.1. To change this address, use the `ioctl`(2) request `SIOCSIFADDR`.

SEE ALSO

`inet`(4P)

NAME

`log`, `klog` – System message log files

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `/dev/log` and `/dev/klog` special files contain messages for the `syslogd`(8) system log daemon. The `/dev/log` file is the user-level system log; it contains the messages issued by `syslog`(3C). The `/dev/log` file is a FIFO special file (named pipe).

The `/dev/klog` file is the kernel-level system log; it contains the kernel messages from the system log daemon, `syslogd`(8). The `/dev/klog` file is a circular queue; the kernel writes kernel messages into it, and `syslogd`(8) reads kernel messages from it.

NOTES

Only the `syslogd`(8) utility should read `/dev/log` and `/dev/klog`. When two or more processes have `/dev/log` or `/dev/klog` open at the same time, results are undefined.

FILES

`/dev/klog`	Kernel-level system log
`/dev/log`	User-level system log

SEE ALSO

`logger`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`syslog`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`syslogd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`mdd` – Mirrored disk driver

IMPLEMENTATION

Cray PVP systems with an IOS model E

DESCRIPTION

The files in `/dev/mdd` are character special files that allow read and write operations to mirrored disk slices. A *mirrored disk slice* is a logical disk device composed of two or more physical disk slices. These physical slices, also known as *members*, must be of the same length and physical sector size. A maximum of `MDDSLMAX` (defined in the parameter file) mirrored devices can exist.

A mirrored disk device takes on the physical sector size of its member physical disk devices. If the sector size of the member devices consists of more than 512 words, the I/O request lengths and starts must match those for the specified device.

Device driver-level mirroring is used when reliability is a critical issue. An individual write I/O request is sent to all members of the mirror that are write enabled. Read I/O requests are sent to the first read-enabled device of the mirrored group that is not busy.

On read errors, the driver selects the next available device from which to read.

On write errors, the member in error is disabled, and no reads or writes go to that device. The node also is marked showing that the device is now out of sync with the other mirrored members. This marking is recorded externally in the `/dev/mdd/`*name* node for the device, permitting the information to be preserved across reboots. To resynchronize, use the `mddconf`(8) and `mddcp`(8) commands.

The files in `/dev/mdd` are not usually mountable as file systems. You may specify a mirrored disk slice as a whole or part of a logical disk device. The files in `/dev/mdd` are all of the *logical indirect* type. See `dsk`(4), `ldesc`(5), `ldd`(4), and `pdd`(4).

The `mknod` command is used as follows to create a mirrored disk inode:

`mknod` *name type major minor* `0` *rwmode path*

name	Name of the logical device.
type	Type of the device data being transferred. Devices in `/dev/mdd` are character devices denoted by a `c`.
major	Major device number of the mirrored logical disk device driver. The driver is denoted by the name `dev_mdd` in the `/usr/src/uts/c1/cf/devsw.c` file.
minor	Minor device number for this slice. Each striped disk slice must have a unique minor device number.
`0`	Placeholder for future use.

rwmode Read/write/initialize modes, in the form `0`*xx*. Reading from right to left, each digit represents a member of mirrored group. The bits represent read enable, write enable, and initialize, and they also are read from left to right, as permissions on a file are read. For example, 037 is a two-member mirror: member 0 is read/write/initialize; member 1 is write only and initialize.

path Path name that designates the logical descriptor file listing the member slices. See `ldesc`(5).

NOTES

As noted above, an unsynchronized mirrored device is marked in the `/dev/mdd/`*name* node. If these nodes are recreated, causing the loss of synchronization information, the device must be resynchronized manually.

EXAMPLES

The following example creates a mirrored disk inode:

```
mknod /dev/mdd/usr c dev_mdd 2 0 077 /dev/ldd/usr.mirror
```

FILES

`/dev/mdd/*`

`/usr/include/sys/mdd.h`

`/usr/src/c1/io/mdd.c`

SEE ALSO

`dsk`(4), `ldd`(4), `ldesc`(5), `pdd`(4)

`mddconf`(8), `mddcp`(8), `mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`mem`, `kmem` – Common or main memory files

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `mem` file is a special file that is an image of the common or main memory of the computer. It can be used to examine or patch the system. References to nonexistent locations fail with an `errno` of `ENXIO`.

The `kmem` file is the same as `mem`; `kmem` has been preserved, because physical memory and kernel memory are not exactly the same on some machines not manufactured by Cray Research.

You also can use the `mem` (or `kmem`) file to change the configuration of physical memory. Physical memory is made up of the following areas:

Kernel memory	Three separate areas that are not necessarily contiguous: • Space allocated at build time • Space allocated at boot time • Space allocated at run time
RAM disk	Memory-resident disk.
User memory	Area in which user processes reside.
Guest memory	Area in which guest operating systems reside (mutually exclusive from downed memory).
Downed memory	Area that was formerly used by diagnostics. This area is an artifact of the archaic form of `chmem` (mutually exclusive from guest memory).
Maintenance memory	Area used by diagnostics.

The following `ioctl`(2) requests are defined in the `sys/mm.h` include file:

`MM_STATUS_ONLY`	Returns memory status.
`MM_RSV_MAINT`	Reserves maintenance memory by reducing the overall size of configured physical memory by the amount required for diagnostics; the required space comes from user memory. On completion, if requested, the memory status is returned.
`MM_RLS_MAINT`	Releases maintenance memory by restoring the overall size of configured physical memory; the space used for maintenance memory is returned back to user memory. On completion, if requested, the memory status is returned.

The value of the `ioctl` argument *arg* is the address of the following structure, which is defined in `sys/mm.h`:

```
struct mmioctl {
        struct mmstat  *mmi_stat;       /* memory status area (user
                                           relative) */
        int             mmi_statlen;    /* length of mem status area
                                           (bytes) */
};
```

If `mmi_statlen` is 0, no memory status will be returned.

Memory status (that is, the current configuration of physical memory) is returned by using the following structure. A `ba` suffix refers to base address, and an `sz` suffix refers to size. All units are in words.

```
struct mmstat {
        long    mms_flags;      /* memory state flags */
        long    mms_cnfphyssz;  /* configured physical memory size*/
        long    mms_actphyssz;  /* actual physical memory size */
        long    mms_kbuildba;   /* kernel allocated at build time */
        long    mms_kbuildsz;
        long    mms_kbootba;    /* kernel allocated at boot time */
        long    mms_kbootsz;
        long    mms_krunba;     /* kernel allocated at run time
                                   (i.e., kmem) */
        long    mms_krunsz;
        long    mms_ramba;      /* ramdisk */
        long    mms_ramsz;
        long    mms_usrba;      /* user memory */
        long    mms_usrsz;
        long    mms_plockdsz;   /* amount of user memory that’s
                                   plocked */
        long    mms_downedba;   /* downed memory */
        long    mms_downedsz;
        long    mms_maintba;    /* maintenance memory */
        long    mms_maintsz;
};
```

In addition to the standard `ioctl` error codes (see `ioctl`(2)), the following are errors that cause an `ioctl`(2) request to fail:

`EACCES`	Calling process was not plocked in memory when trying (possibly indirectly) to reduce the size of user memory
`EAGAIN`	Maintenance memory is disabled while downed or guest memory exists
`EBUSY`	Cannot idle system (that is, park all CPUs) so that memory can be reconfigured

`EDOM`	User’s and kernel’s idea of memory status structure size differ
`EFAULT`	Cannot copy information from or to user area
`EINTR`	Interrupted system call
`EINVAL`	Unrecognized request or device minor number is not 0, 1, or 2
`ENOSPC`	Space for maintenance memory is unavailable from user memory
`ENOSYS`	Maintenance memory is not supported

FILES

```
/dev/MAKE.DEV
/dev/kmem
/dev/mem
```

NAME

`mnu` – Interactive `mnu`-based display package

SYNOPSIS

`/usr/src/uts/cmd/disk/mnu.c`

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The `mnu` program is a display package that provides a menu-based interactive command processing capability. An application program that uses the `mnu` program provides a set of linked menus in a structured form and is compiled with `mnu`. The application calls a function in `mnu` to update the display and to solicit input from users.

The `mnu` program provides command input and screen refresh capabilities by using the following primitives:

`<TAB>` or right arrow	Menu right
`<BACK SPACE>` or left arrow	Menu left
`<RETURN>`	Next menu level or execute menu item
`<ESC>`	Back to first menu level
`<CONTROL-F>` or `<PAGE DOWN>`	Next display page
`<CONTROL-B>` or `<PAGE UP>`	Previous display page
`<CONTROL-D>` or down arrow	Display down one line
`<CONTROL-U>` or up arrow	Display up one line
`?`	Help

The first letter of a given menu item selects and executes that menu item. A help facility displays help text for each menu item if provided by the application.

An application builds and links together menu items by using the `mnu` structure defined in `/usr/include/sys/mnu.h`.

```
/*
 *  menu structure
 */
struct mnu {
    char            *mu_name;       /* menu name */
    struct mnu      *mu_forw;       /* next menu level */
    int             (*mu_func)();   /* function to execute */
    int             mu_flags;       /* flags defined below */
    char            *mu_help;       /* pointer to help text */
};

/*
 *  menu flags
 */
#define MUF_INPUT   1               /* input mode */
#define MUF_STAY    2               /* stay on this menu */
```

The mu_name field is the name of the menu item. The mu_forw field points to the next level menu structure and is NULL at the bottom menu. The mu_func field is a pointer to a function, on the bottom menu item, that performs the desired action. The mu_flags control menu displays action, and the mu_help field is a pointer to optional help text.

The application program must provide an external mnu entry called mnu0 that points to the main (top) menu. The application calls mnu_refresh to update the display and menu items at the specified refresh rate.

The hddmon(8) utility provides an example of an application that makes use of mnu display and command capabilities. See /usr/src/uts/cmd/disk/hddmon.c.

FILES

/usr/src/uts/cmd/disk/hddmon.c

/usr/src/uts/cmd/disk/mnu.c

/usr/include/sys/mnu.h

SEE ALSO

hddmon(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`etc/netconfig` – Network configuration database

IMPLEMENTATION

Cray Research systems licensed for ONC+^{TM} and UNICOS 8.3 or later

DESCRIPTION

The network configuration database, `/etc/netconfig`, is a system file used to store information about networks that are connected to the system. The network selection component also includes the `NETPATH` environment variable and a group of routines that access the network configuration database by using `NETPATH` components as links to the `netconfig` entries.

The `netconfig` database contains an entry for each network available on the system. Entries are separated by newlines. Fields are separated by white space in a prescribed order. You can embed white space as a blank single space or a `tab` symbol. You may embed backslashes (\) as symbols. Lines in `/etc/netconfig` that begin with a # symbol in column 1 are treated as comments.

Each of the valid lines in the `netconfig` database correspond to an available transport. Each entry is of the following form and order:

- *network ID*
- *semantics*
- *flag*
- *protocol family*
- *protocol name*
- *network device*
- *translation libraries*

network ID	A string that uniquely identifies a network. *network ID* consists of nonnull characters, and it has a length of at least 1. No maximum length is specified. This namespace is locally significant and is named by the local system administrator. All network IDs on a system must be unique.
semantics	The *semantics* field is a mandatory field that contains a string that identifies the semantics of a network. Semantics is defined as the services a network supports and the service interface the network provides. The following semantics are recognized.

`tpi_clts`	Transport Provider Interface (connectionless).
`tpi_cots`	Transport Provider Interface (connection oriented).
`tpi_cots_ord`	Transport Provider Interface (connection oriented) it supports an orderly release.

flag The *flag* field records two-valued (true and false) attributes of networks. *flag* is a string composed of a combination of symbols, each of which indicates the value of the corresponding attribute. If a specified symbol is present, the attribute is true. If a symbol is absent, the attribute is false. The `-` symbol indicates that none of the attributes are present. Only one symbol is currently recognized:

`v`	Visible (default) network. Used when the `NETPATH` environment variable is not set.

protocol family The *protocol family* and *protocol name* fields are provided for protocol-specific applications. The *protocol family* field contains a string that identifies a protocol family. The *protocol family* identifier follows the same rules as those for network IDs; the string consists of nonnull characters, it has a length of at least 1, and no maximum length is specified. A `-` symbol in the *protocol family* field indicates that no protocol family identifier applies (the network is experimental). The following are examples:

`loopback`	Loopback (local to host)
`inet`	Internetwork: UDP, TCP, and so on

protocol name The *protocol name* field contains a string that identifies a protocol. The *protocol name* identifier follows the same rules as those for network IDs; that is, the string consists of nonnull characters, it has a length of at least 1, and no maximum length is specified. A `-` symbol indicates that none of the names listed apply. The following protocol names are recognized.

`tcp`	Transmission Control Protocol
`udp`	User Datagram Protocol

network device The *network device* field is the full path name of the device used to connect to the transport provider. The following network devices are recognized.

`/dev/tcp`	Transmission Control Protocol
`/dev/udp`	User Datagram Protocol

translation libraries
The name-to-address *translation libraries* field support a name-to-address mapping service and directory service for the network. A `-` in this field indicates the absence of any translation libraries, in which case, name-to-address mapping for the network is nonfunctional. This field consists of a comma-separated list of path names to libraries. Although this is not used in the UNICOS software, the path should be present.

Each field corresponds to an element in the `struct netconfig` structure. `struct netconfig` and the identifiers described previously are defined in <`netconfig.h`>. This structure includes the following members:

`char *nc_netid`	Network ID, including null terminator
`unsigned long nc_semantics`	Semantics
`unsigned long nc_flag`	Flags

`char *nc_protofmly`	Protocol family
`char *nc_proto`	Protocol name
`char *nc_device`	Full path name of the network device
`unsigned long nc_nlookups`	Number of directory lookup libraries
`char **nc_lookups`	Names of the name-to-address translation libraries
`unsigned long nc_unused[9]`	Reserved for future expansion

The `nc_semantics` field takes the following values, corresponding to the semantics identified previously:

- `NC_TPI_CLTS`
- `NC_TPI_COTS`
- `NC_TPI_COTS_ORD`

The `nc_flag` field is a bit field. The `NC_VISIBLE` bit, corresponding to the attribute identified previously, is currently recognized.

`NC_NOFLAG` indicates the absence of any attributes.

WARNINGS

You should not modify the `/etc/netconfig` file provided by Cray Research, because incoherent behavior of `rpcbind`(8) may result.

FILES

`netconfig.h`

SEE ALSO

`nsswitch`(4)

`rpcbind`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`nis` – A new version of the network information service

IMPLEMENTATION

Cray Research systems licensed for ONC+™ and UNICOS 8.3 or later

DESCRIPTION

NIS+ is a new version of the network information service. This version differs in several significant ways from version 2, which is referred to as NIS or YP in earlier releases. Specific areas of enhancement have been the ability to scale to larger networks, security, and the administration of the service.

The man pages for NIS+ are broken up into two basic categories. Section 8 man pages are user commands and daemons. Section 3N man pages describe the NIS+ programming API.

All commands and functions that use NIS version 2 are prefixed by the letters `yp` as in `ypcat`(1). Commands and functions that use the new version are prefixed by the letters `nis`, as in `nismatch`(8) and `nis_add_entry`(3N).

This man page introduces NIS+ terminology. It also describes the NIS+ namespace and authentication and authorization policies.

NIS+ Namespace

The naming model of NIS+ is based on a tree structure. Each node in the tree corresponds to an NIS+ object. There are six types of NIS+ objects:

- Directory
- Table
- Group
- Link
- Entry
- Private

NIS+ directories

Each NIS+ namespace will have at least one NIS+ directory object. An NIS+ directory is like a UNIX file system directory that contains other NIS+ objects, including NIS+ directories. The NIS+ directory that forms the root of the NIS+ namespace is called the *root directory*. Two special NIS+ directories exist: `org_dir` and `groups_dir`. The `org_dir` directory consists of all systemwide administration databases, such as `passwd`, `hosts`, and `groups`. The `groups_dir` directory consists of NIS+ group objects that are used for access control. The collection of the `org_dir`, `groups_dir` and their parent directory is referred to as an *NIS+ domain*. You can arrange NIS+ directories in a tree-like structure allowing the NIS+ namespace to be divided so that it matches an organizational hierarchy.

NIS+ tables

NIS+ tables, contained within NIS+ directories, store the actual information about some particular type. For example, the hosts system table stores information about the IP address of the hosts in that domain. NIS+ tables have multiple columns and any of the columns can be searched. Each table object defines the schema for its database.

NIS+ group objects

NIS+ group objects are used for access control at group granularity. NIS+ group objects, contained within the `group_dir` directory of a domain, contain a list of all NIS+ principals within a certain NIS+ group.

NIS+ link objects

NIS+ link objects are similar to UNIX symbolic file system links. Typically, they are used for short cuts in the NIS+ namespace.

For more information about the NIS+ objects, see `nis_objects`(3N).

NIS+ entry objects

The NIS+ tables consist of NIS+ entry objects. For each entry in the NIS+ table, an NIS+ entry object exists. NIS+ entry objects conform to the schema defined by the NIS+ table object.

Multiple Administrative Domains

NIS+ allows the creation of multiple domains, or subset of the enterprise network, that may be administered on an autonomous basis. As a corporation grows, or as its corresponding domain grows, authorized administrators can subdivide the domain into two or more hierarchical subdomains.

NIS+ allows for a primary copy of information to be stored on a master server, with zero or more slave servers storing replicas of the primary copy. Updates are made only to the master server, which propagates them to its slave servers. An NIS+ client can send look-up requests to any of the replicas and update requests only to the master server. This arrangement has two benefits: inconsistent updates between tables is avoided because only one master exists, and either a master or a slave server can act as a back-up server for look-up requests.

Each domain in an NIS+ network has its own master server and also may have many slave replicated servers. The overall reliability of the network is enhanced when multiple master servers are across the network, one for each domain, as opposed to one master domain for an NIS+ network. If a master server is down, only updates for its particular domain are disabled; updates to the rest of the network are not affected.

NIS+ Names

The NIS+ service defines two forms of names, simple names and indexed names. The service uses simple names to identify NIS+ objects contained within the NIS+ namespace. Indexed names are used to identify NIS+ entries contained within NIS+ tables. Entries within NIS+ tables also are returned to the caller as NIS+ objects of type entry. NIS+ objects are implemented as a union structure, which is described in the <`rpcsvc/nis.h`> file. The `nis_objects`(3N) man page describes the differences between the various types and the components of these objects.

Simple Names

Simple names are made up of a series of labels that are separated by the dot (`.`) symbol. Each label is composed of printable symbols from the ISO Latin 1 set. Each label can be of any nonzero length, provided that the fully qualified name is fewer than `NIS_MAXNAMELEN` octets, including the separating dots. For the actual value of `NIS_MAXNAMELEN`, see the <`rpcsvc/nis.h`> header file. You must use quotation marks for labels that contain special symbols. See the Grammar subsection.

The NIS+ namespace is organized as an individual rooted tree. Simple names identify nodes within this tree. These names are constructed such that the leftmost label in a name identifies the leaf node, and all of the labels to the right of the leaf identify that objects parent node. The parent node is referred to as the *leafs directory* (this is a naming directory and should not be confused with a file system directory).

For example, the name `example.simple.name` is a simple name that has three labels:

`example`	The leaf node in this name
`simple`	The directory of this leaf
`name`	Simple name of the directory

The `nis_leaf_of`(3N) function returns the first label of a simple name. The `nis_domain_of`(3N) function returns the name of the directory that contains the leaf. Repeated use of these two functions can break a simple name into each of its label components.

The name dot (`.`) is reserved to name the global root of the namespace. For systems that are connected to the Internet, this global root will be served by a domain name service (DNS). When an NIS+ server is serving a root directory whose name is not dot (`.`), this directory is referred to as a *local root*. The root of the NIS+ namespace does not have to be the local root, and it ends in a trailing dot.

NIS+ names are said to be fully qualified when the name includes all of the labels that identify all of the directories, up to the global root. Names without the trailing dot are called *partially qualified*.

Indexed Names

Indexed names are compound names that are composed of a search criterion and a simple name. The search criterion component is used to select entries from a table; the simple name component is used to identify the NIS+ table that will be searched. The search criterion is a series of column names and their desired values enclosed in bracket (`[]`) symbols. These criteria take the following form:

```
[column_name=value, column_name=value, ... ]
```

A search criterion is combined with a simple name to form an indexed name by concatenating the two parts, separated by a comma (`,`) symbol, as follows.

```
[ search-criterion ],table.directory.
```

When multiple column name/value pairs are present in the search criterion, only those entries in the table that have the appropriate value in all columns specified are returned. When no column name/value pairs are specified in the search criterion, all entries in the table are returned.

Grammar

The following text represents a context-free grammar that defines the set of legal NIS+ names. The terminals in this grammar are the following symbols:

- Dot (`.`)
- Open bracket (`[`)
- Close bracket (`]`)
- Comma (`,`)
- Equals (`=`)
- White space

Angle brackets (`<` and `>`), which delineate nonterminals, are not part of the grammar. The vertical bar (`|`) symbol is used to separate alternate productions, and it should be read as either this production or the following production:

name	`::= . |` <*simple name*> `|` <*indexed name*>
simple name	`::=` <*string*>`. |` <*string*>`.`<*simple name*>
indexed name	`::=` <*search criterion*>`,`<*simple name*>
search criterion	`::= [` <*attribute list*> `]`
attribute list	`::=` <*attribute*> `|` <*attribute*>`,`<*attribute list*>
attribute	`::=` <*string*> `=` <*string*>
string	`::= ISO Latin 1 character set`

The `/` symbol is not used. The initial character may not be a terminal character or the symbols at (`@`), plus (`+`), or hyphen(-).

Terminals that appear in strings must be quoted with double quotation marks (`"`). You may quote the `"` symbol by quoting it with itself (`""`).

Name Expansion

The NIS+ service accepts only fully qualified names. Because such names may be unwieldy, however, the NIS+ commands use a set of standard expansion rules that will try to fully qualify a partially qualified name. The NIS+ library function `nis_getnames`(3N) actually does this expansion. This function generates a list of names by using the default NIS+ directory search path or the `NIS_PATH` environment variable. The default NIS+ directory search path includes all of the names in its path. When the `EXPAND_NAME` flag is used, the `nis_lookup`(3N) and `nis_list`(3N) functions invoke `nis_getnames`(3N).

The `NIS_PATH` environment variable contains an ordered list of simple names. The names are separated by the `:` symbol. If any name in the list contains colons, you should quote the colon as described in the Grammar subsection. When the list is exhausted, the resolution function returns the error `NIS_NOTFOUND`. This may end up masking the fact that the name existed but a server for it was unreachable. If the name presented to the list or look-up interface is fully qualified, the `EXPAND_NAME` flag is ignored.

In the list of names from the `NIS_PATH` environment variable, the `$` symbol is treated specially. Simple names that end with the `$` have this symbol replaced by the default directory. For more information, see `nis_local_directory`(3N). Using the `$` as a name in this list results in this name being replaced by the list of directories between the default directory and the global root that contain at least two labels.

An example of this expansion follows. If the default directory is a long name (such as `some.long.domain.name.`), and the `NIS_PATH` variable is set to `fred.bar.:org_dir.$:$`, this path is initially broken up into the following list:

1. `fred.bar.`
2. `org_dir.$`
3. `$`

The `$` in the second component is replaced by the default directory. The `$` in the third component is replaced with the names of the directories between the default directory and the global root that have at least two labels in them. The effective path value becomes:

1. `fred.bar.`
2. `org_dir.some.long.domain.name.`
3. `some.long.domain.name.`
4. `long.domain.name.`
5. `domain.name.`

Each of these simple names is appended to the partially qualified name that was passed to the `nis_lookup`(3N) or `nis_list`(3N) interface. Each is tried until `NIS_SUCCESS` is returned or the list is exhausted.

If the `NIS_PATH` variable is not set, the path `$` is used.

The `nis_getnames`(3N) function may be called from user programs to generate the list of names that would be searched. You also can use the `nisdefaults`(8) program with the `-s` option to show the fully expanded path.

Concatenation Path

Usually, all of the entries for a certain type of information are stored within the table itself. At times, however, it is desirable for the table to point to other tables where entries can be found. For example, you may want to store all IP addresses in the host table for their own domain, and yet want to be able to resolve hosts in some other domain without explicitly specifying the new domain name. With a concatenation path, you can create a sort of flat namespace out of a hierarchical structure. You also can create a table with no entries and just point the hosts or any other table to its parent domain. With such a set up, you are moving the administrative burden of managing the tables to the parent domain. The concatenation path slows down the request response time because more tables and more servers are searched.

NIS+ provides a mechanism for concatenating different but related tables with a "NIS+ Concatenation Path." This path is set up at table creation time by using the `nistbladm`(8) command. You can specify more than one table to be concatenated, and they are searched in the given order. The NIS+ client libraries will not follow the concatenation path set in the other tables.

Namespaces

The NIS+ service defines two additional disjoint namespaces for its own use. These namespaces are the NIS+ Principal namespace and the NIS+ Group namespace. The names associated with the group and principal namespaces are syntactically identical to simple names. However, the information they represent cannot be obtained by directly presenting these names to the NIS+ interfaces. Special interfaces are defined to map these names into NIS+ names so that they may then be resolved.

Principal Names

NIS+ principal names uniquely identify users and machines that are making NIS+ requests. These names have the following form:

principal.domain

The *domain* is the fully qualified name of an NIS+ directory in which the specified principals credentials can be found. For more information on domains, see the Directories and Domains subsections. No leaf exists in the NIS+ namespace in the name, *principal*.

Credentials are used to map the identity of a host or user from one context such as a process UID into the NIS+ context. They are stored as records in an NIS+ table named `cred`. `cred` is always found in the `org_dir` subdirectory of the directory specified in the principal name.

You can express this mapping as a replacement function, as follows:

principal.domain `->[cname=`*principal.domain*`],cred.org_dir.domain`

This latter name is an NIS+ name that can be presented to the `nis_list`(3N) interface for resolution. To administer the NIS+ principal names, use the `nisaddcred`(8) command.

The `cred` table contains the following five columns:

- `cname`
- `auth_name`
- `auth_type`
- `public_data`
- `private_data`

One record in this table exists for each identity mapping for an NIS+ principal. The current service supports two such mappings:

`LOCAL`	Maps from the UID of a given process to the NIS+ principal name associated with that UID. If no mapping exists, the name `nobody` is returned. When the effective UID of the process is 0 (for example, the privileged user), the NIS+ name associated with the host is returned. UIDs are sensitive to the context of the machine on which the process is executing.
`DES`	Maps to and from a Secure RPC netname into an NIS+ principal name. Because netnames contain the notion of a domain, they span NIS+ directories.

The NIS+ client library function `nis_local_principal`(3N) uses the `cred.org_dir` table to map the UNIX notion of an identity, a process UID, into an NIS+ principal name. Shell programs can use the command `nisdefaults`(8) with the `-p` option to return this information.

To map from UIDs to an NIS+ principal name, construct a query in the following form:

`[auth_type=LOCAL, auth_name=`*uid*`],cred.org_dir.`*defaultdomain*`.`

This query returns a record that contains the NIS+ principal name associated with this UID in the machines default domain.

The NIS+ service uses DES mapping to map the names associated with Secure RPC requests into NIS+ principal names. RPC requests that use Secure RPC include the netname of the client making the request in the RPC header. This netname has the following form:

`unix.`*UID@domain*

The service constructs a query by using the following form:

`[auth_type=DES, auth_name=`*netname*`],cred.org_dir.`*domain.*

The domain part is extracted from the netname, rather than using the default domain. This query is used to look up the mapping of this *netname* into an NIS+ principal name in the domain in which it was created.

This mechanism of mapping UID and network names into an NIS+ principal name ensures that a client of the NIS+ service has only one principal name. This principal name is used as the basis for authorization, which is described as follows. All objects in the NIS+ namespace and all entries in NIS+ tables must have an owner specified for them. This owner field always contains an NIS+ principal name.

Group Names

Like NIS+ principal names, NIS+ group names take the form:

group_name.domain

All objects in the NIS+ namespace and all entries in NIS+ tables may optionally have a group owner specified for them. This group owner field, when filled in, always contains the fully qualified NIS+ group name.

The NIS+ client library defines several interfaces for dealing with NIS+ groups. For information on these interfaces, see the `nis_groups`(3N) man page. These interfaces internally map NIS+ group names into an NIS+ simple name that identifies the NIS+ group object associated with that group name. This mapping looks like the following:

group.domain `->` *group.groups_dir.domain*

This mapping eliminates collisions between NIS+ group names and NIS+ directory names. For example, without this mapping, a directory with the name `engineering.foo.com.` would make it impossible to have a group named `engineering.foo.com.`. This is due to the restriction that within the NIS+ namespace, a name unambiguously identifies one object. With this mapping, the NIS+ group name `engineering.foo.com.` maps to the NIS+ object name `engineering.groups_dir.foo.com.`.

The contents of a group object is a list of NIS+ principal names, and the names of other NIS+ groups. For a more complete description of their use, see `nis_groups`(3N).

Directories and Domains

Some directories within the NIS+ namespace are referred to as NIS+ Domains. Domains are those NIS+ directories that contain the `groups_dir` and `org_dir` subdirectories. The `org_dir` subdirectory should contain the table named `cred`. In particular, because of the way the group namespace and the principal namespace are implemented within the NIS+ namespace, NIS+ Group names and NIS+ Principal names always include the NIS+ domain name after their first label.

NIS+ Security

Unlike NIS, NIS+ defines a security model to control access to information managed by the service. The service defines access rights that are selectively granted to individual clients or groups of clients. Principal names and group names are used to define clients and groups of clients that may be granted or denied access to NIS+ information.

The security model also uses the notion of a class of principals called `nobody` that contains all clients, whether or not they have authenticated themselves to the service and the class world. The class world includes any client who has been authenticated.

Authorization

The NIS+ service defines the following four access rights that can be granted or denied to clients of the service:

- `read`
- `modify`
- `create`
- `destroy`

These rights are specified in the object structure at creation time and may be modified later by using the `nischmod`(8) command. Generally, the rights granted for an object apply only to that object. However, for purposes of authorization, rights granted to clients reading directory and table objects are granted to those clients for all of the objects contained by the parent object. This notion of containment is abstract. The objects do not actually contain other objects within them. Group objects do contain the list of principals within their definition.

Access rights are interpreted as follows:

`read`	This right grants read access to an object. For directory and table objects, having read access on the parent object conveys read access to all of the objects that are direct children of a directory, or entries within a table.
`modify`	This right grants modification access to an existing object. Read access is not required for modification. In many applications, however, you must read an object before modifying it. Such modify operations will fail unless you also grant read access.
`create`	This right gives a client permission to create new objects where one had not previously existed. It is used only in conjunction with directory and table objects. Create access for a table allows a client to add additional entries to the table. Create access for a directory allows a client to add new objects to an NIS+ directory.
`destroy`	This right gives a client permission to destroy or remove an existing object or entry. When a client tries to destroy an entry or object by removing it, the service first checks to see whether the table or directory containing that object grants the client destroy access. If it does, the operation proceeds, if the containing object does not grant this right, the object itself is checked to see whether it grants this right to the client. If the object grants the right, the operation proceeds; otherwise, the request is rejected.

Each of these rights may be granted to any one of four different categories, as follows:

`owner`	A right may be granted to the owner of an object. The owner is the NIS+ principal identified in the owner field. To change the owner, use the `nischown`(8) command. If the owner does not have modification access rights to the object, the owner cannot change any access rights to the object, unless the owner has modification access rights to its parent object.
`group owner`	A right may be granted to the group owner of an object. This grants the right to any principal that is identified as a member of the group associated with the object. To change the group owner, use the `nischgrp`(8) command. The object owner does not have to be a member of this group.
`world`	A right may be granted to everyone in the world. This grants the right to all clients who have authenticated themselves with the service.
`nobody`	A right may be granted to the `nobody` principal. This has the effect of granting the right to any client that makes a request of the service regardless of whether or not they are authenticated.

For bootstrapping reasons, directory objects that are NIS+ domains, the `org_dir` subdirectory, and the `cred` table within that subdirectory must have `read` access to the `nobody` principal. This makes navigation of the namespace possible when a client is in the process of locating its credentials. Granting this access does not allow the contents of other tables (such as the password table) within `org_dir` to be read.

Directory authorization

Additional capabilities are provided for granting access rights to clients for directories. These rights are contained within the object access rights (OAR) structure of the directory. This structure allows the NIS+ service to grant rights that are not granted by the directory object to be granted for objects contained by the directory of a specific type.

An example of this capability is a directory object that does not grant create access to all clients, but does grant create access in the OAR structure for group type objects to clients who are members of the NIS+ group associated with the directory. In this example, the only objects that could be created as children of the directory would have to be of the `type` group.

Another example is a directory object that grants `create` access only to the owner of the directory, and additionally grants `create` access through the OAR structure for objects' types (for example, table, link, group, and private) to any member of the directories group. This OAR structure allows complete `create` access to a group except for creating subdirectories. This also restricts the creation of new NIS+ domains, because creating a domain requires creating both a `groups_dir` and `org_dir` subdirectory.

Currently, no command-line interface exists to set or change the object access rights of the directory object.

Table authorization

As with directories, additional capabilities are provided for granting access to entries within tables. Rights granted to a client by the access rights field in a table object apply to the table object and all of the entry objects contained by that table. If an access right is not granted by the table object, it may be granted by an entry within the table. This holds for all rights except `create`.

For example, a table may not grant `read` access to a client performing a `nis_list`(3N) operation on the table. However, the access rights field of entries within that table may grant `read` access to the client. Access rights in an entry are granted to the owner and group owner of the entry and not the owner or group of the table. When the list operation is performed, all entries to which the client has `read` access are returned. Those entries that do not grant `read` access are not returned. If none of the entries that match the search criterion grant `read` access to the client making the request, no entries are returned and the result status contains the `NIS_NOTFOUND` error code.

Access rights that are granted by the *rights* field in an entry are granted for the entire entry. In the table object, however, an additional set of access rights is maintained for each column in the table. These rights apply to the equivalent column in the entry. The rights are used to grant access when neither the table nor the entry itself grants access. The access rights in a column specification apply to the owner and group owner of the entry, rather than the owner and group owner of the table object.

When a `read` operation is performed, if `read` access is not granted by the table and is not granted by the entry but is granted by the access rights in a column, that entry is returned with the correct values in all columns that are readable and the string `*NP*` in columns in which `read` access is not granted.

As an example, consider a client that has performed a list operation on a table that does not grant read access to that client. Each entry object that satisfied the search criterion specified by the client is examined to see whether it grants `read` access to the client. If it does, it is included in the returned result. If it does not, each column is checked to see whether it grants `read` access to the client. If any columns grant `read` access to the client, data in those columns is returned. Columns that do not grant `read` access have their contents replaced by the string `*NP*`. If none of the columns grant `read` access, then the entry is not returned.

FILES

All clients of the NIS+ service should include the `rpcsvc/nis.h` header file.

SEE ALSO

`ypcat`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`nis_groups`(3N), `nis_local_names`(3N), `nis_names`(3N), `nis_objects`(3N), `nis_subr`(3N), `nis_tables`(3N) in *ONC+ Technology for the UNICOS Operating System*, Cray Research publication SG–2169

`newkey`(8), `nisaddcred`(8), `nischown`(8), `nisdefaults`(8), `nismatch`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`nisfiles` – NIS+ database files and directory structure

IMPLEMENTATION

Cray Research systems licensed for ONC+™ and UNICOS 8.3 or later

DESCRIPTION

The Network Information Service Plus (NIS+) uses a memory based, replicated database. This database uses a set of files in the `/etc/nis` directory for checkpointing to stable storage and for maintaining a transaction log. The NIS+ server and client also use files in this directory to store binding and state information.

The NIS+ service implements an authentication and authorization system that is built upon Secure RPC. In this implementation, the service uses a table named `cred.org_dir.domainname` to store the public and private keys of principals that are authorized to access the NIS+ namespace. It stores group access information in the subdomain `groups_dir.domainname` as group objects. These two tables appear as files in the `/etc/nis/hostname` directory on the NIS+ server.

Unlike the previous versions of the network information service in NIS+, the information in the tables is initially loaded into the service from the ASCII files on the server and then updated using NIS+ utilities. For details, see the `nistbladm`(8) man page and the `-D` option description.

The following files are stored in the `/etc/nis` directory:

`NIS_COLDSTART`
: This file contains NIS+ directory objects that will be preloaded into the NIS+ cache at start-up time. This file usually is created at NIS+ installation time. For more information, see `nisinit`(8).

`NIS_SHARED_DIRCACHE`
: This file contains the current cache of NIS+ bindings being maintained by the cache manager. To view the contents, use the `nisshowcache`(8) command.

`hostname.log`
: This file contains a transaction log that is maintained by the NIS+ service. To view it, use the `nislog`(8) command. This file contains holes. Its apparent size may be a lot larger than its actual size. There is only one transaction log per server.

`hostname.dict`
: This file is a dictionary that the NIS+ database uses to locate its files. The default NIS+ database package creates the dictionary. The dictionary has no log file.

`hostname`
: This directory contains databases that the server uses.

`hostname/root.object`
: On root servers, this file contains a directory object that describes the root of the namespace.

`hostname/parent.object`
: On root servers, this file contains a directory object that describes the parent namespace. The `nisinit`(8) command creates this file. If this is an isolated namespace, this file is not created.

`hostname`/*table_name*
: For each table in the directory, there will be a file with the same name that stores the information about that table. If subdirectories are within this directory, the database for the table is stored in the file *table_name*.subdirectory.

`hostname`/*table_name*.log
: This file contains the database log for the table *table_name*. The log file maintains the state of individual transactions to each database. When a database has been checkpointed (that is, all changes have been made to the `hostname`/*table_name* stable storage), this log file will have a length of 0.

  Currently, NIS+ does not do checkpointing automatically. Administrators should execute the `nisping`(8) command with the `-C` option once a day to checkpoint the log file. To accomplish this, use either a `cron`(8) job or execute the command manually each time. For more information, see `nisping`(8).

`hostname.root_dir`
: On root servers, this file stores the database associated with the `root` directory. It is similar to other table databases. The corresponding log file is called `root_dir.log`.

`hostname/cred.org_dir`
: This table contains the credentials of principals in this NIS+ domain.

`hostname/groups_dir`
: This table contains the group authorization objects that NIS+ needs to authorize group access.

NOTES

Except for the `NIS_COLDSTART` and the `NIS_SHARED_DIRCACHE` file, no other files should be manipulated by commands such as `cp`(1), `mv`(1), or `rm`(1). Because the transaction log file keeps logs of all changes made, you must not manipulate the files independently.

SEE ALSO

`nis`(4)

`nis_db`(3N), `nis_objects`(3N) in *ONC+ Technology for the UNICOS Operating System*, Cray Research publication SG–2169

`niscat`(8), `nisinit`(8), `nislog`(8), `nismatch`(8), `nisping`(8), `nistbladm`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

np – Network packet driver for low-speed interfaces

IMPLEMENTATION

Cray PVP systems with an IOS model E

DESCRIPTION

The np driver provides an interface for all of the low-speed network devices connected to an IOS model E (IOS-E), including NSC devices such as the N130 and the FEI3 VME interface. The driver accepts standard UNICOS read(2), write(2), open(2), close(2), listio(2), reada(2), and writea(2) system calls; it also accepts ioctl(2) requests.

By convention, the N-packet special file names are in the following format:

/dev/comm/*ioc iop chan*/lp *nn*

ioc	Single-digit I/O cluster number.
iop	Single-digit IOP number.
chan	Two-digit octal channel number.
nn	Low-order 4 bits of the minor device number ("logical path" in IOS terminology).

Usually, logical path 5 (/dev/comm/*/lp05) is reserved for TCP/IP. You also may use other minor devices, depending on how the interfaces are configured. Check with your system support staff for details of usage at your site. The contents of the device node determine the configuration of the communication device; see the Configuration subsection.

You cannot open a minor device when it is already open. Any attempt to do so fails with an EBUSY error. Each device has a maximum of 16 paths. For driver type NP_FLG_RAW, you may open only one path.

ioctl Requests

The ioctl(2) requests described in this section are defined in sys/np.h.

The following ioctl(2) request requires no parameters:

NPC_HLTIO Halts an outstanding I/O request. This is valid only for asynchronous I/O. All synchronous requests will block.

The following ioctl(2) request requires one parameter in the npioctl structure:

NPC_DEVCNTL Network interface device control. The valid values for sfunc are as follows (see epackn.h):

NP_DC_MC	0	Channel master clear
NP_DC_OD	1	Output disconnect
NP_DC_AUTO_OD	2	Set auto disconnect mode in CCA1
NP_DC_CLR_AUTO_OD	3	Clear auto disconnect mode in CCA1

The following `ioctl`(2) requests require parameters in the `npstat` structure. The `npstat` structure is as follows:

```
struct npstat {
        int     sfunc;          /* status request subfunction            */
        char    *sbuf;          /* status buffer pointer                 */
        int     dev;            /* device number (-1 means all)          */
        int     lpath;          /* logical path number (-1 means all)    */
        uint    slen;           /* status buffer length (bytes)          */
        uint    epoch;          /* incremented every configuration change */
};
```

`NPC_CDSTATS` Clears device statistics request. This is a super-user-only request. The user passes the address of an `npstat` structure by using the `ioctl`(2) request. The requested (all, if `dev` = `-1`) device statistics are cleared. The `epoch` variable is returned in the `npstat` structure.

`NPC_CLSTATS` Clears logical path statistics request. This is a super-user-only request. The user passes the address of an `npstat` structure by using the `ioctl`(2) request. The requested (all, if `lpath` = `-1`) path statistics are cleared. The `epoch` variable is returned in the `npstat` structure.

`NPC_DEV_STATUS` Network logical channel statistics request. This is a super-user-only request. The channel statistics are placed in a buffer specified in the request. The structures that follow define the statistics.

The first five words of `NPC_DEV_STATUS` are status from the configure up request. The first word is channel status:

```
uint    nsr_ibz:  1,    /* input channel busy flag  */
        nsr_idn:  1,    /* input channel done flag  */
                 14,
        nsr_ics: 16,    /* input channel status     */
        nsr_obz:  1,    /* output channel busy flag */
        nsr_odn:  1,    /* output channel done flag */
                 14,
        nsr_ocs: 16;    /* output channel status    */
```

The next four words of `NPC_DEV_STATUS` are valid only for N130 devices. If the master clear request was successful, the response is the "Initialize Device Response Parameter Block" that the device sends to the IOS; otherwise, the response is the response to the failing master clear request.

Each open logical path has the last read/write reply trailer from the IOS added to the buffer. The path is identified by one integer that contains the path number. The next word contains the last error from the IOS for that path. The read/write reply trailer is defined as follows:

```
struct np_rw_rep {                  /* read/write reply trailer     */
        uint    nprw_bz  :1,        /* busy flag                    */
                nprw_dn  :1,        /* done flag                    */
                         :14,       /* unused                       */
                nprw_mdc :16,       /* messages discarded (channel) */
                nprw_mdp :16,       /* messages discarded (lchan)   */
                nprw_mbp :16;       /* messages buffered on lchan   */
        uint             :32,       /* unused                       */
                nprw_iob_addr :32;  /* I/O buffer address           */
};

struct np_rw_reps {     /* read/write reply trailer stats  (optional) */
        uint    nprw_status[4];     /* device status                   */
};
```

These structures are replicated for read and write for each logical path:

```
struct np_rw_stats {
        struct np_rw_rep np_rdstats;    /* read statistics             */
        struct np_rw_reps np_rd_dstats; /* read device statistics      */
        struct np_rw_rep np_wrstats;    /* write statistics            */
        struct np_rw_reps np_wr_dstats; /* write device statistics     */
};
```

The entire structure is defined in `sys/np.h`. The supporting structures are in `epackn.h` and `npsys.h`.

`NPC_DSTATUS` Network interface status request. This is a super-user-only request. This request returns statistics from all channel-related activity on the IOS. The user passes the address of an `npstat` structure by using the `ioctl`(2) request. The length of data and the `epoch` variable are returned in the `npstat` structure.

`NPC_ECHO` N-packet echo. This `ioctl`(2) request lets a super user send echo packets to a communications driver in an IOP. The IOP returns the packets; this function may be used to verify and time the request/response path of an N-packet through the IOS.

`NPC_LSTAT` Last status of interface. This request returns the last status from the IOS. If a request returns an error to the user, the `errno` is a generic EIO. This `ioctl`(2) request lets users determine the exact cause of the failure. The errors are defined in `sys/epackn.h`. Recovery from error cases depends on the type of error and the type of interface. Before implementing any recovery techniques, you should thoroughly understand the device and error modes.

`NPC_LSTATUS`	Network logical path status request. This is a super-user-only request. This request returns statistics from all path-related activity on the IOS. The user passes the address of an `npstat` structure by using the `ioctl`(2) request. The length of data and the `epoch` variable are returned in the `npstat` structure.
`NPC_PACKET`	N-packet interface to allow a super user to issue N-packets directly from a user program. This request gives the user complete control of the IOP interface to a specific device.
`NPC_STAT`	Network interface status request. This is a super-user-only request. This request returns statistics from all channel-related activity on the IOS. This is deferred.

Several "driver types" are defined in the configuration. The raw driver allows a process to send data in any format. The other drivers require a network header to be the first words of any data. This header is an NSC message proper (for types MP, PB, and A130).

The message proper is defined as follows:

```
struct mp {
        char  control[2];       /* NSC control word        */
        char  acode[2];         /* NSC access code         */
        char  to[2];            /* NSC destination adapter */
        char  from[2];          /* NSC source adapter      */
        char  param[56];        /* NSC parameters          */
     } ;
```

The `mp` structure is defined in `sys/np.h`. The details of the contents of message proper fields are available in NSC documentation. For most devices such as the VME interfaces, the important fields are the *to* and *from* fields. You can determine the contents of these fields from the network "adapter" addresses of the host computers and the logical path (device minor).

If a `read`(2) system call is not satisfied within the time-out period, an `ELATE` error is returned. If a `write`(2) system call cannot be completed, it may be retried periodically in the time allowed by the time-out period before an `ELATE` error is returned. A `close`(2) system call closes the minor device. Any outstanding system calls are terminated with an EIO error. The time-outs are defined in the N-packet include files, which you should not have to change.

Configuration

The following definitions are used for the `mknod`(8) parameters:

`/etc/mknod` *name* `c` *maj min ioc iop chan cmode dtype dmode adap hwtype*

name	Name of the special file, usually `/dev/comm/`*ioc iop chan* `/lp` *nn*
maj	Major device number, always 35 for the `np.c` driver
min	Minor device number (encoded device and logical path)
ioc	IOS cluster number [0-7] [00-07]

iop	IOS processor number [0, 1, 2, or 3]
chan	IOP channel number in octal [030, 032, 034, or 036]
cmode	Controller mode [0: 6 Mbyte; 1: 12 Mbyte; 2: 12 Mbyte loopback]
dtype	Driver type (see below for *dmode*; meaning depends on `dtype`)

0 Raw driver: *dmode* not used

1 Message proper (FEI-3, Cray-Cray): *dmode* = 0
Message proper (VAXBI): *dmode* = 1

2 Parameter block driver (NSC N130, Ultra LSC)

The following *dmode* bits are valid only for the N130:

dmode bits 0-15 = 0: no special functions in N130
dmode bits 16-31 (`dfunc`) = 0: no driver function in N130
dmode bit 8 - 0400: variable length message propers (mp's) on medium
dmode bit 7 - 0200: CRC (deferred)
dmode bit 6 - 0100: statistics on in N130
dmode bit 5 - 0040: adapter microcode trace on in N130
dmode bit 4 - 0020: send disconnect after parameter block (N130)
dmode bit 3 - 0010: disable write response parameter block (N130)
dmode bit 2 - 0004: DXU master clear at power up (N130)
dmode bit 1 - 0002: purge all network data (N130)
dmode bit 0 - 0001: clear interface only (N130)

4 NSC message proper (A130, CNT LANlord) : *dmode* n/u

dmode	Driver function and driver mode (high-order 16 bits is driver function). Currently, only the PB driver uses the driver function to specify microcode trace modes.
adap	A130/N130 adapter address (in hexadecimal)
hwtype	Code that specifies the type of hardware or adapter on the channel; for a detailed list of codes, see `netdev.h`. Only monitoring software uses this code.

0102	FEI-3
0104	FEI-CN
0105	FEI-DS
0106	FEI-UC
0107	FEI-VA
0110	FEI-VB
0111	FEI-VM
0301	A130 HYPERchannel

0302	N130 HYPERchannel
0303	EN643 Ethernet
0304	DX4130 FDDI
0401	VAXBI
0502	Ultra LSC

NOTES

A flag is required in the inode for control devices for monitors, configuration commands, and so on. This flag disables input, output, and the process of incrementing the `epoch` variable when the device is opened and closed.

WARNINGS

Differences exist between the interfaces supported by this driver and those supported by previous IOS drivers. Do not use include files from `hy`(4) with this interface.

You can open only one logical path for type RAW.

MESSAGES

The driver returns the following error codes:

`EBUSY`	The device special file is currently in use. The device is type RAW, and another path is in use.
`EFAULT`	An `ioctl`(2) request did not have enough buffer space for the data returned.
`ENOTTY`	An attempt was made to close a logical path that was not open.
`ENXIO`	Device special file has a bad channel number. Device special file has a bad minor number. Network device structures are all in use. Network logical path structures are all in use. Device special file has 6-Mbyte set for an N130 device. Attempt was made to close a device that was not open. Attempt was made to close a logical path that was not allocated. Attempt was made to execute an `ioctl`(2) command to issue an device that was not open. Attempt was made to perform an unknown `ioctl` code. `NPC_DEVCNTL ioctl`(2) request had a bad function code or was not from a super user. `NPC_ECHO ioctl`(2) request was not from a super user. `NPC_PACKET ioctl`(2) request was not from a super user.

The driver writes the following error messages to the system log:

```
ERROR: np.c: Cannot allocate device structure
ERROR: np.c: Cannot allocate logical channel
WARNING: np.c: npstrat: unknown driver type
WARNING: np.c: N-packet structure in use.
WARNING: np.c: npintr: no bp structure.
INFO: np.c: path closed - packet returned
INFO: np.c: No free logical devices
INFO: np.c: No free device structures
np.c: output sequence error %x %x
np ioc %d IOP %d ch 0%o error %d (0%o)
np.c: receive pkt sequence error %x %x
np.c: np device structure closed - reopening
npl already in use
```

For IOS-reported errors, the following message is logged:

```
np ioc %d IOP %d ch 0%o error %d (0%o)
```

The driver returns the following error codes:

Value	**Definition**
10	Device-detected error
11	Parity error
12	`SECDED` error
20	Retry of failing request unsuccessful

The following are software detected errors; execution attempted and failed:

Value	**Definition**
100	Local memory not available.
101	IOB memory not available.
102	Driver terminated.
103	Overrun on read request of N-packet; returned data is truncated.
104	CCA-1 hardware information is not valid.
105	CCA-1 input channel time-out.
106	CCA-1 output channel time-out.
107	Halt I/O request.
110	Maximum consecutive errors encountered; driver terminated.
111	Transferred fewer parcels than requested.
112	Cannot create a required IOS-E activity.

113	A parameter block that was not valid was received off of the CCA-a input channel
114	Read request packet time-out.
115	Cannot drain input channel completely at initialization.
116	Cannot buffer entire input; input truncated.
117	Request aborted due to `CLOSE PATH` request.
120	`RELMEM` request failed.
121	Cannot terminate (`TERM`) all driver activities as desired.
122	Microcode in device not supported.
123	Cannot halt I/O in a driver activity as desired.
250	Bad parameter on `TIMER` call.
251	Attempt to start `TIMER` that is already active.
252	Attempt to stop a `TIMER` that is not active.
260	Target memory I/O error: bad channel buffer ordinal.
261	Target memory I/O error: bad transfer direction.
262	Target memory I/O error: bad channel buffer address.
263	Target memory I/O error: hardware error on HISP channel.
264	Target memory I/O error: target memory not available.
265	Target memory I/O error: bad word length parameter.
270	Local memory to or from channel buffer error: bad buffer ordinal.
271	Local memory to or from channel buffer error: bad transfer direction.
272	Local memory to or from channel buffer error: bad buffer address.
273	Local memory to or from channel buffer error: hardware error on I/O.
274	Local memory to or from channel buffer error: bad word length parameter.
277	IOS-E internal error.

The following are parameter errors in the request packet; execution not tried:

Value	**Definition**
300	Packet type is not valid
301	Request code is not valid
302	Channel not configured up
303	Channel number is not valid
304	Channel already configured up

305	No connection path open for this logical path
306	Logical path is not valid
307	Logical path already open
310	CCA-1 mode is not valid
311	Driver type is not valid
312	Driver mode is not valid
313	Requested transfer length is not valid
314	Subfunction is not valid
315	Requested information not available
316	Packet length is not valid
317	Time-out value is not valid (must be nonzero)
320	Initialization of channel pair already in progress
321	Termination of channel pair already in progress
322	Second `OPEN PATH` request on CCA-1-Raw channel
323	Combination of driver type and driver mode is not valid
324	A130 driver not available in hybrid system
325	PB driver mode "input disc after PB" not available
377	Bad packet type (issued by monitor only)

EXAMPLES

This example makes the devices for the following configuration:

device 0: FEI-3 on cluster 3, IOP 1, channel 030, 6-Mbyte mode, logical paths 0 through 7
device 1: N130 on cluster 0, IOP 0, channel 032, (12-Mbyte mode), logical paths 0 through 7
device 2: FEI-1 on cluster 1, IOP 0, channel 036, 6-Mbyte mode, IBM MVS
device 3: FEI-1 on cluster 2, IOP 1, channel 034, 6-Mbyte mode, VAX on port A

```
/etc/mkdir /dev/comm
cd /dev/comm
/etc/mkdir v31-30 n00-32 f10-36 f21-34
cd v31-30
/etc/mknod lp00 c 35 000 3 1 030 0 1 0 0 0102
/etc/mknod lp01 c 35 001 3 1 030 0 1 0 0 0102
/etc/mknod lp02 c 35 002 3 1 030 0 1 0 0 0102
/etc/mknod lp03 c 35 003 3 1 030 0 1 0 0 0102
/etc/mknod lp04 c 35 004 3 1 030 0 1 0 0 0102
/etc/mknod lp05 c 35 005 3 1 030 0 1 0 0 0102
/etc/mknod lp06 c 35 006 3 1 030 0 1 0 0 0102
/etc/mknod lp07 c 35 007 3 1 030 0 1 0 0 0102
cd ../n00-32
/etc/mknod lp00 c 35 020 0 0 032 1 2 0 0 0302
/etc/mknod lp01 c 35 021 0 0 032 1 2 0 0 0302
/etc/mknod lp02 c 35 022 0 0 032 1 2 0 0 0302
/etc/mknod lp03 c 35 023 0 0 032 1 2 0 0 0302
/etc/mknod lp04 c 35 024 0 0 032 1 2 0 0 0302
/etc/mknod lp05 c 35 025 0 0 032 1 2 0 0 0302
/etc/mknod lp06 c 35 026 0 0 032 1 2 0 0 0302
/etc/mknod lp07 c 35 027 0 0 032 1 2 0 0 0302
cd ../f10-36
/etc/mknod lp00 c 35 040 1 0 036 0 3 0 0 0
cd ../f21-34
/etc/mknod lp00 c 35 060 2 1 034 0 3 1 0 0107
```

TCP/IP Configuration Example

Configuration for TCP/IP is done as described previously, except that the device names are `/dev/comm/tcp` *nnnn*; *nnnn* is an octal number, as follows:

The low-order 4 bits of the minor device number are the logical path. The high-order bits are the TCP/IP device number. Therefore, the following is true:

np0 has minors 0 through 17
np1 has minors 20 through 37
np2 has minors 40 through 57
np3 has minors 60 through 77
np4 has minors 100 through 117
np5 has minors 120 through 137
np6 has minors 140 through 157
np7 has minors 160 through 177, and so on

The next example makes the TCP/IP devices for the following configuration:

np0: FEI-3 on cluster 0, IOP 0, channel 030, 6-Mbyte mode, logical path 5
np1: N130 on cluster 0, IOP 0, channel 032, 12-Mbyte mode, logical path 5
np2: Cray channel on cluster 1, IOP 0, channel 036, 6-Mbyte mode, logical path 3
np3: VAXBI on cluster 2, IOP 1, channel 034, 12-Mbyte mode, logical path 5

```
/etc/mkdir /dev/comm
cd /dev/comm
/etc/mknod tcp0005 c 35 005
/etc/mknod tcp0025 c 35 025
/etc/mknod tcp0043 c 35 043
/etc/mknod tcp0065 c 35 065
```

FILES

`/dev/comm/*`	Device special files
`/usr/include/sys/np.h`	Definitions of constants and structures
`/usr/include/sys/npsys.h`	Definitions of constants and structures

SEE ALSO

`fei`(4), `hy`(4), `vme`(4)

`ioctl`(2), `listio`(2), `read`(2), `reada`(2), `write`(2), `writea`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

npcntl – N-packet control interface

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The N-packet control driver provides an interface for controlling the on/off status of NSC HYPERchannel adapters connected to the I/O subsystem (IOS). The N-packet control driver also is used for Cray Research front-end interfaces (FEIs) and VME interfaces attached to the IOS. For more information, see fei(4) and vme(4). The driver accepts standard UNICOS open(2) and close(2) system calls; it also accepts ioctl(2) requests.

The N-packet control driver is represented by the /dev/npcntl special file. Only the super user can use the device, because turning the network channels on and off interrupts network traffic.

You can use the ioctl request NPFC_CONFCHN to change an N-packet channel status in the IOS. The ioctl structure is defined in the sys/npcntl.h include file.

```
struct npc_cntrl {
        int     channel;  /* channel to change */
        int     ios;      /* ios to change     */
        int     state;    /* on/off status     */
        int     mode;     /* channel mode      */
};
```

channel	The N-packet channel that is changing state.
ios	The IOS to which the channel is connected.
state	The state (either 0 (off) or 1 (on)) to which the channel is changing.
mode	The mode in which to initiate the channel. The default mode is NSC; all other modes are deferred.

FILES

/dev/npcntl

/usr/include/sys/npcntl.h

SEE ALSO

fei(4), hy(4), vme(4)

ioctl(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`nsswitch` – Configuration file for the name-service switch

IMPLEMENTATION

Cray Research systems licensed for ONC+™ and UNICOS 8.3 or later

DESCRIPTION

The operating system uses many databases of information about users, groups, and so forth. Data for some of these databases come from a variety of sources. These sources and their lookup-order can be specified in the `/etc/nsswitch.conf` file.

The following databases use the switch:

Database	**Used By**
`automount`	`automount`(8)
`group`	`getgrent`(3C)
`passwd`	`getpwent`(3C)
`protocols`	`getprotobyname`(3C)
`publickey`	`getpublickey`(3R)
`rpc`	`getrpcbyname`(3C)
`services`	`getservbyname`(3C)
`netgroup`	`netgroup`(5)

You may use the following sources:

Source	**Uses**
`files`	`/etc/passwd`
`nis`	NIS (YP)
`nisplus`	NIS+

An entry in `/etc/nsswitch.conf` exists for each database. Typically, these entries will be simple, such as the following:

`protocols: files` or `networks: files nisplus`.

When you specify multiple sources, you may have to define precisely the circumstances under which each source will be tried.

A source returns one of the following status codes:

Status	**Meaning**
`SUCCESS`	Requested database entry was found
`UNAVAIL`	Source is not responding or corrupted
`NOTFOUND`	Source responded `no such entry`
`TRYAGAIN`	Source is busy, might respond to retries

For each status code, two actions are possible:

Action	**Meaning**
`continue`	Try the next source in the list
`return`	Return now

The complete syntax of an entry follows:

```
<entry>      k::= <database> : [<source> [<criteria>]]* <source>
<criteria>   ::= [ <criterion>+ ]
<criterion>  ::= <status> = <action>
<status>     ::=success | notfound | unavail | tryagain
<action>     ::= return | continue
```

Each entry occupies one line in the file. Lines that are blank or that start with # symbol or with white space are ignored. The <*database*> and <*source*> names are case-sensitive, but <*action*> and <*status*> names are case-insensitive.

The library routines contain default entries that are used if the appropriate entry in `nsswitch.conf` is absent or syntactically incorrect.

The default criteria is to continue on anything except `SUCCESS`; that is, [`SUCCESS=return NOTFOUND=continue UNAVAIL=continue TRYAGAIN=continue].`

The default, or explicitly specified, criteria is meaningless following the last source in an entry, and it is ignored because the action is always to return to the caller regardless of the status code that the source returns.

Interaction with NIS+ and YP-compatibility Mode

The NIS+ server can be run in YP-compatibility mode. When you specify this mode, the server handles NIS (YP) requests and NIS+ requests. The results are the same, except that the `getpwent`(3C) routine uses the `nis` source rather than `nisplus`. You should use the `nisplus` source rather than the `nis` source.

Useful Configurations

The default entries for all databases use NIS+ as the enterprise level name-service. They are identical to those in the default configuration of this file:

Category	Entry
`passwd:`	`files nisplus`
`group:`	`files nisplus`
`protocols:`	`nisplus [NOTFOUND=return] files`
`rpc:`	`nisplus [NOTFOUND=return] files`
`ethers:`	`nisplus [NOTFOUND=return] files`
`publickey:`	`nisplus [NOTFOUND=return] files`
`automount:`	`files nisplus`
`services:`	`nisplus [NOTFOUND=return] files`

The policy `nisplus [NOTFOUND=return] files` implies that if `nisplus` is unavailable, continue on to `files`; if `nisplus` returns `NOTFOUND`, return to the caller. That is, treat `nis` as the authoritative source of information and try `files` only if `nisplus` is down.

NOTES

Within each process that uses `nsswitch.conf`, the entire file is read only once; if the file is changed later, the process will continue using the old configuration.

You should not use both `nis` and `nisplus` as sources for the same database because both name services are expected to store similar information and the lookups on the database may yield different results, depending on which name-service is operational at the time of the request.

Misspelled names of sources and databases will be treated as legitimate names of nonexistent sources and databases.

FILES

`/etc/nsswitch.conf`	Configuration file
`/etc/nsswitch.files`	Sample configuration file that uses only `files`
`/etc/nsswitch.nis`	Sample configuration file that uses `files` and `nis`
`/etc/nsswitch.nisplus`	Sample configuration file that uses `files` and `nisplus`

SEE ALSO

`netconfig`(4), `nis`(4), `ypfiles`(5) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`automount`(8), `ifconfig`(8), `nisd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`null` – Null file

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `/dev/null` file is a character special file. Data written on `/dev/null` is discarded; read operations from `/dev/null` always return 0 bytes.

FILES

`/dev/MAKE.DEV`

`/dev/null`

SEE ALSO

`mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`pdd` – Physical disk device interface

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The files in `/dev/pdd` are special files that allow read and write operations to physical disk devices. Each file represents one slice of a physical disk device. The files in `/dev/pdd` are character special files that may be used directly to read and write physical disk slices. Usually, they are called to perform I/O on behalf of higher-level logical disk device drivers. For I/O on a character disk device, read and write operations must transfer multiples of the physical device sector size and all seek operations must be on physical sector size boundaries.

The files in `/dev/pdd` are not usually mountable as file systems, although you may combine one or more physical disk slices to make a mountable logical disk device (see `dsk`(4), `ldd`(4), and `mount`(8)).

The files in `/dev/pdd` are created by using the `mknod` command (see `mknod`(8)). Each must have a unique minor device number, along with other parameters used to define a physical disk slice.

The `mknod`(8) command for physical disk devices is as follows:

`mknod` *name type major minor dtype iopath start length flags altpath unit*

name	Descriptive file name for the device (for example, `pdd/scr0230`).
type	Type of the device data being transferred. Devices in `/dev/pdd` are character devices denoted by a `c`.
major	Major device number for physical disk devices. The `dev_pdd` name label in the `/usr/src/uts/c1/cf/devsw.c` file denotes the major device number for physical disk devices. You can specify the major number as `dev_pdd`.
minor	Minor device number for this slice.
dtype	Physical disk device types are defined in `/usr/src/uts/c1/sys/pddtypes.h`. Supported physical disk device types are as follows:

```
    #define DD49    3       /* DD49 disk drive  */
    #define DD40    6       /* DD40 disk drive  */
    #define DD50    7       /* DD50 disk drive  */
    #define DD41    9       /* DD41 disk drive  */
    #define DD60    10      /* DD60 disk drive  */
    #define DD61    11      /* DD61 disk drive  */
    #define DD62    12      /* DD62 disk drive  */
    #define DD42    13      /* DD42 disk drive  */
    #define DA62    14      /* DA62 disk drive  */
    #define DA60    15      /* DA60 disk drive  */
    #define DD301   16      /* DD301 disk drive */
    #define DA301   17      /* DA301 disk drive */
    #define DD302   18      /* DD302 disk drive */
    #define DA302   19      /* DA302 disk drive */
```

CRAY EL series disk types:

```
    #define DDESDI  64      /* old esdi drive */
    #define DD3     65      /* new esdi drive */
    #define DDLDAS  66      /* old Max Strat DAS */
    #define DDAS2   67      /* new Max Strat DAS */
    #define DD4     68      /* ipi + sabre 7 */
    #define RD1     69      /* removable esdi */
    #define DDIMEM  70      /* ironics+memory vme boards */
    #define DD5S    71      /* DD5S SCSI drive */
    #define DD5I    72      /* DD5I IPI drive */
```

iopath The *iopath* specifies the I/O cluster, the I/O processor (IOP), and the controller channel number. For example, an *iopath* of 01234 is IOC 1, IOP 2, channel 34. The *iopath*, defined by the `io_path` structure in `sys/pdd.h`, follows. The structure is different for CRAY EL series, CRAY J90 series, and Cray PVP systems with an IOS model E. The unit is not used here; it is in a separate field, described below.

```
/*
 *  i/o path to the channel adapter
 */
struct io_path {
#if defined(CRAYEL)
        uint            :32,            /* must remain unused */
                unit    :8,
                ioc     :8,             /* ios - vme backplane */
                iop     :8,             /* eiop - controller   */
                chan    :8;             /* channel             */
#elif defined(CRAYJ90)
        uint            :32,            /* must remain unused */
                unit    :8,
                ioc     :6,             /* ios - vme backplane */
                iop     :9,             /* eiop - controller   */
                chan    :9;             /* channel             */
#else
        uint            :32,            /* unused */
                unit    :16,            /* unit */
                        :3,             /* unused */
                ioc     :4,             /* io cluster */
                iop     :3,             /* io processor */
                chan    :6;             /* channel */
#endif /* CRAYELS */
};
```

start Absolute starting block (sector) number of the slice.

length Number of blocks (sectors) in the slice.

flags Flags for physical disk device control, defined in `sys/eslice.h`, follows. They are mainly used for diagnostic and maintenance purposes. Usually, the flags field should be 0 for slices in `/dev/pdd`.

```
#define S_CONTROL       001    /* control device              */
#define S_NOBBF         002    /* no bad block forwarding     */
#define S_NOERREC       004    /* no error recovery           */
#define S_NOLOG         010    /* no error logging            */
#define S_NOWRITEB      020    /* no write behind             */
#define S_CWE           040    /* control device write enable */
#define S_NOSPIRAL      0100   /* no spiraling                */
```

altpath The optional alternate *iopath* that you can use as a back-up path to the physical disk device's second port.

unit The disk device unit number for device types that support multiple units on the same channel.

`ioctl` Requests

The physical disk driver supports the following `ioctl`(2) requests. They are defined in `sys/pddtypes.h`, and they are passed as the *cmd* argument in the `ioctl`(2) system call.

If the `ioctl` description indicates that the `ioctl`(2) request has no effect on CRAY EL series systems, the call may be part of a command that the system supports, but the call has no meaning on the system. If the description indicates that the `ioctl`(2) request is not supported on CRAY EL series systems, the request is used only by commands not supported on CRAY EL series systems.

`PDI_STOP`	(Not supported on the CRAY EL series) Stops the queued disk requests after all outstanding requests finish.
`PDI_START`	(Not supported on the CRAY EL series) Resumes disk requests after they are stopped by using a `PDI_STOP ioctl`(2) request.
`PDI_DOWN`	Puts device in a down state and terminates all queued requests with an error.
`PDI_UP`	Puts device in an up state.
`PDI_RDONLY`	Sets device to a read-only state.
`PDI_NOALLOC`	Sets device to a state in which writes can occur but no new file allocation can take place. The file system uses this request.
`PDI_SPINIT`	(Not supported on the CRAY EL series) Initializes the spare sector map for the specified device.
`PDI_DIAG_REQ`	(Not supported on the CRAY EL series) Registers the calling process for a diagnostic function request. A `read`(2) or a `write`(2) system call by the calling process at a later time is treated as a diagnostic request. The argument is a pointer to a disk request packet, `drq_pak`, defined in `sys/epackd.h`.
`PDI_DIAG_RES`	(Not supported on the CRAY EL series) Registers the calling process for a diagnostic function response. An IOS response to a `read`(2) or a `write`(2) system call by the calling process at a later time is copied into the caller. The argument is a pointer to a disk response packet to which the response is copied. The disk response packet, `drs_pak`, is defined in `sys/epackd.h`.
`PDI_GETFLAGS`	(Not supported on the CRAY EL series) Copies the physical device control flags to the word to which *arg* points. The device control flags are defined previously.
`PDI_SETFLAGS`	(Not supported on the CRAY EL series) Sets the physical device control flags to the contents of *arg*. The device control flags are defined previously.
`PDI_SPIN_UP`	(Has no effect on the CRAY EL series) Issues a spin-up function to the physical device. The device must have this capability and be in remote mode.
`PDI_SPIN_DOWN`	(Has no effect on the CRAY EL series) Issues a spin-down function to the physical device. The device must have this capability and be in remote mode.
`PDI_GETMODE`	Gets the current read/write mode.

`PDI_GETSTATE`	Gets the current disk state.
`PDI_PRIMARY`	Selects primary path to disk.
`PDI_ALTERNATE`	(Has no effect on the CRAY EL series) Selects alternate path to disk.
`PDI_RESET`	Resets device stats.
`PDI_GET_STREAMS`	(Has no effect on the CRAY EL series) Gets streams.
`PDI_SET_STREAMS`	(Has no effect on the CRAY EL series) Sets streams.
`PDI_GET_SL_STREAMS`	(Has no effect on the CRAY EL series) Gets slice streams.
`PDI_SET_SL_STREAMS`	(Has no effect on the CRAY EL series) Sets slice streams.
`PDI_ATOM_CP`	(Not supported on the CRAY EL series) Atomic read/write diagnostic function.
`PDI_RESYNC`	(Has no effect on the CRAY EL series) Resyncs labels to spindle within array.
`PDI_LDFRMT`	(Not supported on the CRAY EL series) Loads format spec to spindle within array.

EXAMPLES

The following `mknod`(8) command makes a node for `pdd/scr0230`, type `c`, major number `dev_pdd`, minor number 110, disk type DD-60, I/O cluster 0, IOP 2, channel 30, startg at block 0, length of 1472 blocks, 0 for flags, no alternate path, and unit number of 1:

```
mknod pdd/scr0230 c dev_pdd 110 10 0230 0 1472 0 0 1
```

FILES

```
/dev/pdd/*
/usr/include/sys/pdd.h
/usr/include/sys/pddprof.h
/usr/src/c1/io/pdd.c
```

SEE ALSO

`dsk`(4), `ldd`(4), `mdd`(4), `sdd`(4), `ssdd`(4)

`ddstat`(8), `mknod`(8), `mount`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

proc – Process file system

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /proc file system allows users access to the address space of a running process. This file system consists of files named /proc/*nnnnn*; *nnnnn* is the process ID formatted in decimal. Each file contains the address space of the process it represents.

Access to each member of /proc is restricted by the typical file system protection mechanisms, with the additional restrictions that operations such as chown(1) and chmod(1) are prohibited in the /proc file system. The /proc file system does not have an associated device driver, but a major device number is still needed for its current operation.

To inspect or modify the address space of a process by using the /proc file system, open the file in /proc that represents that process by using the open(2) system call, and then use the lseek(2), read(2), or write(2) system call to access the process's address space.

Additional operations on processes opened through /proc are supported by the ioctl(2) system call, allowing debuggers to control the execution of the subject process precisely and to obtain a file descriptor that refers to the subject process's text file. The /proc ioctl Requests subsection describes all /proc ioctl(2) operations.

Configuration

To create a /proc file system, use the following steps:

1. Create an empty directory called /proc by using the following command:

   ```
   mkdir /proc
   ```

2. Modify the system mount scripts and /etc/fstab file so that the /proc file system is always mounted at system startup. For the format of the /proc entry in the fstab file, see fstab(5).

Process Address Space Segmentation

When a process is opened through the /proc file system, its address space is segmented into several distinct address spaces. These separate process address spaces are declared in the sys/fs/prfcntl.h include file and are defined as follows:

PRFS_DATA	Program data space.
PRFS_TEXT	Program text space.
PRFS_PREGS	Primary registers, including the process's P register, A and S registers, and the VL and VM registers. The structure of this address space is in the proc_pregs structure as defined in the sys/fs/prfcntl.h include file.

`PRFS_VREGS`	Process V registers. The eight vector registers of the process appear in sequential order in this address space. Thus, the first 64 words (512 bytes) in this address space correspond to V0, and the second group of 64 words corresponds to V1.
`PRFS_BREGS`	Process B registers. The 64 B registers of the process appear in sequential order in this address space. Thus, the word at byte offset 0 in this address space corresponds to B0; the word at byte offset 8 corresponds to B1.
`PRFS_TREGS`	Process T registers. The 64 T registers of the process appear in sequential order in this address space. Thus, the word at byte offset 0 in this address space corresponds to T0; the word at byte offset 8 corresponds to T1.
`PRFS_SM`	Process shared semaphores. This address space is exactly 1 word in length with the low-order bits of the word defining the state of the shared semaphores. The number of shared semaphores varies by machine architecture: 32 semaphores for CRAY Y-MP systems. This address space is read-only.
`PRFS_SHRDBREGS`	Process shared SB registers. The shared SB registers of the process appear in sequential order in this address space. Thus, the word at byte offset 0 in this address space corresponds to SB0; the word at byte offset 8 corresponds to SB1.
`PRFS_SHRDTREGS`	Process shared ST registers. The shared ST registers of the process appear in sequential order in this address space. Thus, the word at byte offset 0 in this address space corresponds to ST0; the word at byte offset 8 corresponds to ST1.

The following process address spaces are included for convenience (because they reference internal UNICOS data structures, compatibility across releases is not supported):

`PRFS_PCOMM`	Process common structure as defined in the `sys/proc.h` include file. This address space is read-only.
`PRFS_PROC`	Process structure as defined in the `sys/proc.h` include file. This address space is read-only.
`PRFS_SESS`	Session table structure as defined in the `sys/session.h` include file. This address space is read-only. If the process is not in a session, any attempted read from this address space will return 0 bytes.
`PRFS_UCOMM`	User common structure as defined in the `sys/user.h` include file. This address space is read-only.
`PRFS_USER`	User structure as defined in the `sys/user.h` include file. This address space is read-only.

The method used to access a particular location in any of the address spaces is always the same. If the current position of the file on which a process is open is not already at the proper location, an `lseek`(2) system call should be made that identifies both the address space and the beginning byte offset within the given address space, followed by a `read`(2) or `write`(2) system call to access the data. (Alternatively, you can use the `listio`(2) system to perform both the seek and read or write operation in one system call.)

For example, the following code fragment reads the first element of the second vector register (V1) of the process open on the `fd` file descriptor:

```
#include <sys/fs/prfcntl.h>
                :
         long   buf;
                :
         lseek (fd, PRFS_VREGS | 64*sizeof(long), 0);
         read (fd, (char *)&buf, sizeof(long) );
```

Splitting the process address space into multiple discontinuous segments results in some slightly peculiar behavior because one I/O operation on a `/proc` file is not permitted to cross a segment boundary. Thus, I/O operations that run beyond the end of a segment are truncated.

The `/proc` files differ from other UNICOS files because various portions of the address space are always read-only (for example, `PRFS_PROC`); other UNICOS files are either entirely writable or entirely write-protected.

/proc ioctl Requests

The format for `ioctl`(2) requests to `/proc` is as follows:

```
#include <sys/fs/prfcntl.h>
ioctl (fildes, request, arg)
long *arg;
```

The valid `ioctl`(2) requests are as follows:

`PFCCSIG`	If the *arg* argument is set to 0, clears all pending signals; otherwise, *arg* points to a signal mask that contains the signal numbers to be cleared.
`PFCCSIGM`	If the *arg* argument is set to 0, clears all pending signals for the multitasking group; otherwise, *arg* points to a signal mask that contains the signal numbers to be cleared.
`PFCEXCLU`	Marks the process text space for exclusive use. The *arg* argument is not used and should be set to 0.
`PFCGMASK`	Gets the signal trace bit mask of the process. The *arg* argument must point to a long integer in which the signal trace bit mask will be returned (see `PFCSMASK`).
`PFCGMASKM`	Gets the signal trace bit mask of the multitask group. The *arg* argument must point to a long integer in which the signal trace bit mask will be returned (see `PFCSMASKM`).
`PFCKILL`	Sends a signal to the process. The *arg* argument must point to a long integer that contains the number of the signal to be sent.

`PFCKILLM`	Sends a signal to all processes of the multitask group. The *arg* argument must point to a long integer that contains the number of the signal to be sent.
`PFCOPENT`	Opens text file for reading. The *arg* argument must point to an integer in which the opened file descriptor referring to the process's text file will be returned.
`PFCREXEC`	Clears the stop-on-exec flag of the process. The *arg* argument is not used and should be set to 0.
`PFCRUN`	Makes the process runnable. The *arg* argument is not used and should be set to 0.
`PFCRUNM`	Makes all processes of the multitask group runnable. The *arg* argument is not used and should be set to 0.
`PFCSEXEC`	Sets the stop-on-exec flag of the process. The *arg* argument is not used and should be set to 0.
`PFCSMASK`	Sets the signal trace bit mask of the process. The *arg* argument must point to a long integer that defines the signal trace bit mask. The process stops when any signal is received whose corresponding bit in the trace mask also is set. The trace bit mask for signal *s* is as follows: `1L << (s-1)`
`PFCSMASKM`	Sets the signal trace bit mask of the multitask group. The *arg* argument must point to a long integer that defines the signal trace bit mask. When any process in the multitask group receives the signal whose corresponding bit in the trace mask also is set, the receiving process stops, and all other processes in the multitask group, are sent the `STOP` signal. The trace bit mask for signal *s* is as follows: `1L << (s-1)`
`PFCSTOP`	Sends the `STOP` signal to the process. The *arg* argument is not used and should be set to 0.
`PFCSTOPM`	Sends the `STOP` signal to all processes in the multitask group of which the process is a member. The *arg* argument is not used and should be set to 0.
`PFCWSTOP`	Waits for the process to become stopped. You can use the *arg* argument as a pointer to an integer to which the status of the stopped process will be returned. Any status value returned in this way is interpreted in a manner identical to the status returned by the `wait`(2) system call. Alternatively, the *arg* argument can be 0, indicating that no status information will be returned.
`PFCWSTOPM`	Waits for all processes in the multitask group to stop. The *arg* argument is not used and should be set to 0.

`PFCQUERYM`	Returns the status of each member in the multitask group. The *arg* argument should be a pointer to a structure of type `struct pfcquery`, which contains a pointer to an array of type `struct pfcstatus` and the size of the array. The status of each task in the multitask group, up to the given maximum, is returned in the array. Each array element contains the process identifier of the task, the status of the task, and some flags. The only flag currently implemented is `PSTAT_UNKNOWN`, which means that the process has neither stopped nor exited.
`PFCSWITCHM`	When debugging a multitask group, there is at any given time, a currently traced task, which is identified by its process identifier. If the *arg* argument is nonzero, it points to the process identifier, a task that is then set to be the currently traced task. Alternatively, the *arg* argument may be 0, which leaves the currently traced task unchanged. The previous currently traced tasks's PID is returned as the function value.

NOTES

The use and implementation of the `ioctl`(2) operations documented in this entry are subject to change in future releases of UNICOS.

FILES

`/proc`	`/proc` file system root
`/usr/include/sys/fs/prfcntl.h`	Definitions for the process address space

SEE ALSO

`fstab`(5)

`close`(2), `ioctl`(2), `listio`(2), `read`(2), `reada`(2), `write`(2), `writea`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`pty` – Pseudo terminal interface

IMPLEMENTATION

CRAY Y-MP systems

CRAY J90 series

CRAY EL series

DESCRIPTION

The pseudo terminal interface, `pty`, provides support for a device pair called a *pseudo terminal*, which is a pair of character devices. This pair consists of a master device and a slave device. The slave device provides an interface for processes that is identical to that described in `termio`(4). However, whereas all other devices that provide the interface described in `termio`(4) have a hardware device of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the pseudo terminal. That is, anything written on the master device is given to the slave device as input, and anything written on the slave device is presented as input on the master device.

The `ioctl` requests that apply to pseudo terminals are defined in the `sys/pty.h` include file, as follows:

`FIONBIO`	Enables or disables nonblocking I/O. Nonblocking I/O is enabled by the specification (by reference) of a nonzero parameter and is disabled by a 0 parameter. When nonblocking I/O is enabled, a read or write operation returns the error `EWOULDBLOCK`, rather than going to sleep to wait for the input buffer to fill or the output buffer to empty.
`TCIOEXT`	Enables or disables external processing mode. External processing allows programs that use pseudo terminals more control over echoing of data.
`TCRDFL`	Enables "daemon read failure" mode. This mode allows the daemon to detect a read request on the master pty device without using the `ioctl` request `TCTTRD`. The daemon's read operation fails with the `ENOMSG` error. This error occurs whenever a read request is on both the pty and tty sides, and no data is going in either direction. When a daemon read fails with `ENOMSG`, the daemon should write before it issues another read request.
`TCSIG`	Sends a signal to the client's process group; the signal sent is specified by the *arg* argument in the `ioctl`(2) request.
`TCTTRD`	Returns a nonzero value if a process on the master pty device currently has an outstanding `read`(2) system call. The address of the word that stores the return value is specified by the *arg* argument in the `ioctl`(2) request.

`TIOCPKT` Enables or disables packet mode. Packet mode is enabled by the specification (by reference) of a nonzero parameter and disabled by a 0 parameter. When this request is applied to the master side of a pseudo terminal, each subsequent read operation from the terminal returns data written on the slave part of the pseudo terminal, preceded by a 0 byte (symbolically defined as `TIOCPKT_DATA`) or a single byte that reflects control status information. In the latter case, the byte is an inclusive OR of 0 or more bits. The symbolic definition of the bytes is as follows:

`TIOCPKT_FLUSHREAD`	Sets whenever the read queue for the terminal is flushed.
`TIOCPKT_FLUSHWRITE`	Sets whenever the write queue for the terminal is flushed.
`TIOCPKT_STOP`	Sets whenever output to the terminal is stopped with <`CONTROL-s`>.
`TIOCPKT_START`	Sets whenever output to the terminal is restarted.
`TIOCPKT_DOSTOP`	Sets whenever `IXON` terminal control mode is enabled (see `termio`(4)).
`TIOCPKT_NOSTOP`	Sets whenever `IXON` terminal control mode is disabled (see `termio`(4)).

The `rlogin`(1B) and `rlogind`(8) commands use packet mode to implement a remote login with remote echoing, local flow control with <`CONTROL-s`> and <`CONTROL-q`>, and proper back-flushing of output. Other similar programs also can use this mode.

BUGS

You cannot send an EOT to a pseudo terminal.

FILES

`/dev/pty/`*nnn*

`/dev/ttyp`*nnn*

`/usr/include/sys/pty.h`

SEE ALSO

`termio`(4), `tty`(4)

NAME

qdd – Physical disk device interface

IMPLEMENTATION

CRAY J90se systems

CRAY T90 systems

DESCRIPTION

The files in `/dev/qdd` are special files that allow read and write operations to physical disk devices connected to the IPN-1. Each file represents one slice of a physical disk device. The files in `/dev/qdd` are character special files that may be used directly to read and write physical disk slices. Usually, they are called to perform I/O on behalf of higher-level logical disk device drivers. For I/O on a character disk device, read and write operations must transfer multiples of the physical device sector size and all seek operations must be on physical sector size boundaries.

The files in `/dev/qdd` are not usually mountable as file systems, although you may combine one or more physical disk slices to make a mountable logical disk device (see `dsk`(4), `ldd`(4), and `mount`(8)).

The files in `/dev/qdd` are created by using the `mknod` command (see `mknod`(8)). Each must have a unique minor device number, along with other parameters used to define a physical disk slice.

The `mknod`(8) command for physical disk devices is as follows:

`mknod` *name type major minor dtype iopath start length flags altpath unit*

name	Descriptive file name for the device (for example, `qdd/scr0230`).
type	Type of the device data being transferred. Devices in `/dev/qdd` are character devices denoted by a `c`.
major	Major device number for physical disk devices. You can specify the major number as `dev_qdd`.
minor	Minor device number for this slice.
dtype	Physical disk device type.
iopath	The *iopath* specifies the GigaRing number the device is on, the node number to which the disk is connected, and the controller and unit number of the device The controller number is in the range 0 through 4. For array devices, the controller number is always 0. The unit number is in the range 0 through 7.
start	Absolute starting block (sector) number of the slice.
length	Number of blocks (sectors) in the slice.
flags	Flags for physical disk device control. They are mainly used for diagnostic and maintenance purposes. Usually, the flags field should be 0 for slices in `/dev/qdd`.

altpath	The optional alternate *iopath* that you can use as a back-up path to the physical disk device's second port.
unit	The disk device unit number for device types that support multiple units on the same channel.

FILES

`/dev/qdd/*`

`/usr/src/c1/io/qdd.c`

SEE ALSO

`dsk`(4), `ldd`(4), `mdd`(4), `sdd`(4), `ssdd`(4) `xdd`(4)

`ddstat`(8), `mknod`(8), `mount`(8), `sdconf`(8), `sdstat`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`ram` – Random-access memory disk interface

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

Random-access memory (RAM) is an area of memory that you may configure as one or more character or block special devices. It is treated as a disk drive from the user level. RAM is configured in `/usr/src/uts/cf/conf.`*SN*`.c` (*SN* is the mainframe serial number); the driver uses the minor device number as an index to a slice description. The RAM interface is represented by the `/dev/ram` special file. The `ramsize` constant determines the amount of memory to be dedicated for all devices in RAM.

EXAMPLES

The following example allocates a total of 200,000 words of main memory to RAM. This example shows the allocation of two devices, each made up of 100,000 words, that could be configured in `/dev` as either character or block special files. Their minor device numbers are 0 and 1; their major device numbers depend on their location in the `bdevsw` or `cdevsw` tables.

```
#define ramsize 200000

struct size ram00[2] = {
        sliceinit(100000,0,0),
        sliceinit(100000,100000,0)
};
```

FILES

`/dev/ram`

`/usr/src/uts/cf/conf.`*SN*`.c` (*SN* is the mainframe serial number)

NAME

`rdd` – RAM disk driver

IMPLEMENTATION

Cray PVP systems with an IOS model E

DESCRIPTION

The files in `/dev/rdd` are character special files that allow read and write operations to random-access memory (RAM) disk slices. Each file represents one slice of the total RAM disk area. The total memory allocated for RAM disks is specified in the UNICOS parameter file.

Usually, I/O request lengths to RAM disks must be in 512-word multiples and start on 512-word boundaries. A RAM disk slice can assume the attributes of a physical disk device. If the sector size of the specified device consists of more than 512 words, the I/O request lengths and starts must match those for the specified device.

Usually, you cannot mount the files in `/dev/rdd` as file systems. You may specify a RAM disk slice as a whole or part of a logical disk device. You also can combine a RAM disk slice with a physical disk device. See `dsk`(4), `ldesc`(5), and `pdd`(4).

The files in `/dev/rdd` are created by using the `mknod`(8) command. Each file must have a unique minor device number, a starting block, and a length (in blocks).

The `mknod`(8) command for RAM disk devices is as follows:

`mknod` *name type major minor dtype* `0` *start length*

name	Descriptive file name for the device.
type	Type of the device data being transferred. Devices in `/dev/rdd` are character devices denoted by a `c`.
major	Major device number for RAM disk devices. The `dev_rdd` name label in the `/uts/c1/cf/devsw.c` file denotes the major device number for RAM disk devices.
minor	Minor device number for this slice. Each RAM disk slice must have a unique minor device number.
dtype	(Optional) Physical disk device type. If left at 0, the RAM disk slice assumes the physical attributes of a DD-49 disk drive. For a list of physical disk device types, see `pdd`(4).
`0`	Placeholder for future use.
start	Absolute starting block (sector) number of the slice.
length	Number of blocks (sectors) in the slice.

FILES

`/dev/rdd/*`

`/usr/src/c1/io/rdd.c`

SEE ALSO

`dsk`(4), `ldd`(4), `ldesc`(5), `pdd`(4)

`mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

reqt – IPI-3 interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /dev/ipi3/reqt device sends the IPI-3/IPI configuration to the IPI-3/IPI packet driver, and it requests configuration, table, and device limit information.

To communicate between the packet driver and the controlling process, use the pki_ctl structure, as defined in the sys/pki_ctl.h include file. The packet driver control structure is defined as follows:

```
struct pki_ctl{
       int    pki_psigno;                 /* Signal to receive        */
       word   *pki_packet;                /* Packet from user program */
       int    pki_nbytes;                 /* Length of packet         */
       int    pki_device;                 /* Device name              */
       }
```

The IPI-3/IPI interface uses the following ioctl(2) requests:

PKI_GET_CONFIG	Returns the IPI-3/IPI configuration
PKI_GET_DEVCONF	Returns the device configuration
PKI_GET_DEVTBL	Returns an IPI-3/IPI table
PKI_GET_OPTIONS	Returns IPI-3/IPI options
PKI_PUT_CONFIG	Sends the IPI-3/IPI configuration
PKI_SET_OPTIONS	Sets the IPI-3/IPI options

FILES

/dev/ipi3/*device-name*	IPI-3/IPI interface devices
/usr/include/sys/pki_ctl.h	Structure definition of pki_ctl

SEE ALSO

ipi3(4)

ipi3_clear(8), ipi3_config(8), ipi3_option(8), ipi3_start(8), ipi3_stat(8), ipi3_stop(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

Tape Subsystem Administration, Cray Research publication SG–2307

NAME

`sdd` – Striped disk driver

IMPLEMENTATION

Cray PVP systems with an IOS model E

DESCRIPTION

The files in `/dev/sdd` are character special files that allow read and write operations to striped disk slices. A *striped disk slice* is a logical disk device composed of two or more physical disk slices. These physical slices, also known as *members*, must be of the same physical device type and length.

Usually, I/O request lengths to striped disks must be in 512-word multiples and start on 512-word boundaries. A striped disk device assumes the physical sector size of its member physical disk devices. If the sector size of the member devices consists of more than 512 words, the I/O request lengths and starts must match those for the specified device.

Device driver level striping is used when an increase in I/O bandwidth is desired. An individual I/O request is divided into component requests, one or more for each member physical device. The basic unit of striped I/O is known as the *stripe factor*. The stripe factor is fixed based on the physical device type of the underlying members.

Usually, you cannot mount the files in `/dev/sdd` as file systems. You can specify a striped disk slice as a whole or part of a logical disk device. The files in `/dev/sdd` are all of the logical indirect type. See `dsk`(4), `ldd`(4), `ldesc`(5), and `pdd`(4).

The `mknod` command is used to create a striped disk inode, as follows:

`mknod` *name type major minor* `0 0` *path*

name	Name of the logical device.
type	Type of the device data being transferred. Devices in `/dev/sdd` are character devices denoted by a `c`.
major	Major device number of the striped logical disk device driver. The name `dev_sdd` in the `/usr/src/uts/c1/cf/devsw.c` file denotes the driver.
minor	Minor device number for this slice. Each striped disk slice must have a unique minor device number.
`0 0`	Placeholders for future use.
path	Path name that designates the logical descriptor file listing the member slices. See `ldesc`(5).

EXAMPLES

The following example creates a striped disk inode:

```
mknod /dev/sdd/usr c dev_sdd 1 0 0 /dev/ldd/usr.stripe
```

FILES

```
/dev/sdd/*
/usr/include/sys/sdd.h
/usr/src/c1/io/sdd.c
```

SEE ALSO

`dsk`(4), `ldd`(4), `ldesc`(5), `pdd`(4)

`mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`sds` – Secondary data storage interface on SSD devices

IMPLEMENTATION

Cray PVP systems (except CRAY J90 series)

DESCRIPTION

The secondary data storage (SDS) device is extended storage space allocated on an SSD (solid-state storage device). It can be used by users for extended storage, or by the operating system for use as logical device cache. See `ssd`(4) and `ldcache`(8).

Users can allocate SDS space using the `ssbreak`(2) system call. Reads and writes of SDS space can use the specialized `ssread`(2) and `sswrite`(2) UNICOS system calls, or the more general purpose `read`(2), `write`(2), `reada`(2), `writea`(2), and `listio`(2) systems calls.

The `ssread`(2) and `sswrite`(2) system calls do not require a file descriptor. There is only one SDS device and only an `ssbreak`(2) system call is required to allocate extended storage before the `ssread`(2) or `sswrite`(2) system calls. The `ssread`(2) and `sswrite`(2) system calls, however, are limited to synchronous operation. See `ssread`(2) and `sswrite`(2).

The character special file, `/dev/sds`, provides a general purpose interface to the SDS device. By opening `/dev/sds`, a file descriptor is obtained to allow `read`(2), `reada`(2), `write`(2), `writea`(2), and `listio`(2) system calls. File permissions on `/dev/sds` allow any user to open it; however, an `ssbreak` is required to allocate space before any reads or writes are allowed.

The character special file, `/dev/sds` is created by the `mknod`(8) command as follows:

```
mknod /dev/sds c dev_sds 0
```

FILES

`/dev/sds`

SEE ALSO

`ssd`(4), `ssdd`(4), `ssdt`(4)

`sdss`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`ssbreak`(2), `ssread`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`ldcache`(8), `ldsync`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`secded` – SECDED maintenance function interface

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The SECDED maintenance functions allow access to the memory error correction interface of the hardware. These functions allow the setting and clearing of data bits or check bits in a word of memory or a processor register. Reading a word that has been set in this way reveals whether memory error detection, correction, and reporting are working properly. The functions also provide a means of controlling the scrubbing of single-bit memory errors. The parameters that control the system's response to bursts of memory errors can be manipulated through this interface.

The `secded` driver supports the `ioctl`(2) and `open`(2) system calls. The driver supports only one device (minor device 0).

The following `ioctl` requests are accepted:

`ME_GET` Accepts as an argument a pointer to a structure that the driver will fill with the memory error correction parameters *mecormax*, maximum number of single-bit errors that can occur in *meint* period of time with no intervals of longer than *meint/mecormax* seconds without an error, then single-bit error detection is turned off for all user processes for *medisint* seconds. The last parameter, *meuncmax*, is the limit of uncorrectable errors that the UNICOS system will allow before forcing a panic because of the memory errors. The structure is defined in `sys/memc.h`. The default parameters are defined by `MECORMAX` as 16 errors, `MEINT` as 5*HZ or 5 seconds, `MEDISINT` as 300 or 60 seconds times 5, and `MEUNCMAX` as 64.

`ME_SET` Accepts as an argument a pointer to a structure that the driver will extract the new memory error correction parameters, *meint, mecormax, medisint,* and *meuncmax*.

`RPE_SET` Accepts as an argument a pointer to a structure that contains a register set designator (`RPE_V`, `RPE_T`, `RPE_B`, `RPE_IB`, or `RPE_SR`) and a parity indicator (even or odd).

`RPE_SET` allows the CPU register parity error functions to be tested. Incorrect even or odd register parity may be written into a V, T, or B register; a shared register; or an instruction buffer, and then read to force a register parity error interrupt.

Special maintenance instructions used for these functions exist only on CRAY Y-MP CPUs that have a revision level of 4 or later. These instructions behave as NO-OPs on other CPUs. Currently, only the V register function (`RPE_V`) is supported. The `RPE_T`, `RPE_B`, `RPE_IB`, and `RPE_SR` functions are deferred.

`SD_SET`	Accepts as an argument a pointer to a structure that contains the address of the word to be modified and the data and check bits to be modified within the word. The `ioctl` request uses the SECDED maintenance instructions to read the entire word (64 data and 8 check bits), complements the data and check bits to be modified, and rewrites the entire 72-bit data word to memory. In this way, any 72-bit pattern, including the associated check bits, can be placed into a memory word.
`CPU_GET`	Accepts as an argument a pointer to a structure that the driver will fill with the parameters controlling downing of CPUs on uncorrectable memory errors: *umemax*, *umelife* and *umedown*. *umemax* is the number of uncorrectable memory errors per CPU that can occur before the CPU is downed by the operating system. Each error has a lifetime of *umelife* seconds, after which it is no longer counted in the number of errors for a CPU. The CPU will remain down for *umedown* seconds, after which it will automatically be returned to service by the operating system. Setting *umemax* to zero disables the automatic downing of the CPU. Setting *umedown* to zero disables the automatic return to service of the CPU. The defaults are zero for *umemax* and *umedown* and 86400 (24 hours) for *umelife*.
`CPU_SET`	Accepts as an argument a pointer to a structure from which the driver will extract the new parameters to control the downing of CPUs on uncorrectable memory errors: *umemax*, *umelife* and *umedown*.

NOTES

For the maintenance functions to work, the error maintenance switch on the mainframe switch panel must be enabled. The software, however, cannot determine whether the switch is enabled. For memory error detection and correction to work properly, the error correction switch on the mainframe also must be on (this is the normal state). Because of the nature of the SECDED maintenance instructions, any test using this capability should be done in single-user mode. The address of a word being modified is checked only to ensure that it is within the physical memory of the machine. If a word is changed to contain a double-bit or multibit error, and the kernel reads that word next (as opposed to a user process reading that word next), the kernel panics.

For CRAY Y-MP systems, you must deadstart the mainframe with the maintenance mode switch in the off position. After the mainframe is deadstarted (and the system is running in single-user mode), you should turn the maintenance mode switch to the on position.

MESSAGES

The following errors can occur:

`EFAULT`	Returned if the word to be modified is outside the machine’s memory, or if the parameter structure is outside the user’s field length.
`ENXIO`	Returned if other than minor device 0 is selected, or if the mainframe is not an appropriate type for the attempted operation.

`EPERM`	Returned if the user is not the super user.
`EINVAL`	Returned if parameters passed in on `SD_SET`, `ME_SET`, `RPE_SET` or `CPU_SET` do not pass validation tests.

FILES

```
/dev/secded
/usr/include/sys/memc.h
/usr/include/sys/secded.h
```

SEE ALSO

`secded`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`sfs` – File that contains the names of each Cray Research system in an SFS cluster and its associated SFS arbiter

SYNOPSIS

`/etc/config/sfs`

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The configuration of SFS arbiters is represented in the `/etc/config/sfs` file. The `/etc/config/sfs` file is generated using the menu system, or created directly with a text editor. The `/etc/config/sfs` file should be identical on all systems within a cluster that share file systems.

The format of the `/etc/config/sfs` file is line oriented. Each line begins with one of two valid keywords starting in column 1: `HostName` or `Arbiter`.

The `HostName` Line

The `HostName` line describes the name of a system, and denotes which SFS arbiters are valid and accessible by that system. For example, the following line defines that the host `frost` is a system within this cluster, and that `frost` can access SFS arbiters 0, 1, and 2, as further defined within `/etc/config/sfs`:

```
HostName        frost   0,1,2
```

The `Arbiter` Line

The `Arbiter` line describes the identity of an SFS arbitration service, and the path names of the three character special devices necessary to support that arbitration service. For example, the following line defines an SFS arbiter with a numeric identity of 0, with a symbolic name of SMP-2, and which uses the three path names `/dev/smp-0`, `/dev/sfs-0`, and `/dev/smnt-0` to access the three character special devices that define an SFS arbiter:

```
Arbiter             0  SMP-2        /dev/smp-0 /dev/sfs-0 /dev/smnt-0
```

The first path name describes the character special for the physical semaphore device. For example, the output of file `/dev/smp-0` may look like the following, which describes an SMP-2 (device type equals 2) attached to low-speed channel pair 18:

```
/dev/smp-0:     character special (73/0)    2 18 0 0 0 0 0 0
```

The output of file `/dev/smp-0` also may look like the following, which describes an H-SMP (device type equals 4) representing port 7, whose HIPPI I/O path is described in the character special node `/dev/hdd/smp`:

```
/dev/smp:       character special (73/0)    4 7 /dev/hdd/smp 0 0
```

The second path name describes the character special for the logical shared file system driver, and its associated Shared Lock Region. For example, the output of file `/dev/sfs-0` may look like the following, which describes an interface to the logical SFS driver that uses `dev/dsk/slr` as the shared medium necessary to communicate semaphore allocation and other shared information to the other systems in the cluster:

```
/dev/sfs-0:     character special (48/0)    0 0 /dev/dsk/slr 0 0
```

The third path name describes the character special for the Shared Mount Table interface. For example, the output of file `/dev/smnt-0` may look like the following, which describes an interface to the logical SFS driver that uses a portion of the shared medium described in `/dev/sfs-0` as a record of SFS mounts to be shared with the other systems in the cluster:

```
/dev/smnt-0:    character special (75/0)    0 0 0 0 0 0 0 0
```

The minor number assigned to each of the three character special devices must be the same, and it must match the SFS arbiter numeric identity defined in the `/etc/config/sfs` file.

EXAMPLES

An example of `/etc/config/sfs` taken from a test system looks like the following:

```
c
o
l
u
m
n

1
|
v
HostName        frost   0,1,2
HostName        ice     0,1,2
HostName        sn5609  1,2
Arbiter         0  SMP-2        /dev/smp-0 /dev/sfs-0 /dev/smnt-0
Arbiter         1  HSMP         /dev/smp-1 /dev/sfs-1 /dev/smnt-1
Arbiter         2  Simulator    /dev/smp-2 /dev/sfs-2 /dev/smnt-2
```

FILES

`/dev/sfs`	External Semaphore Device Logical-layer Interface
`/dev/smp`	Low-level interface to the semaphore device
`/dev/smnt`	Shared mount table interface

SEE ALSO

`esdmon`(8), `sfsd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

Shared File System (SFS) Administrator's Guide, Cray Research publication SG–2114

NAME

`slog` – Security log interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `/dev/slog` pseudo device is a read-only device that holds security log records. The security log daemon, `slogdemon`(8), transfers those records to the security log file for use with the security utilities. For more information about the security log file, see `slrec`(5).

FILES

`/dev/slog`	Security log pseudo device
`/usr/adm/sl/slogfile`	Disk-resident security log

SEE ALSO

`slrec`(5)

`reduce`(8), `slogdemon`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

`ssd` – Solid state storage device

IMPLEMENTATION

Cray PVP systems (except CRAY J90 series)

DESCRIPTION

The SSD solid-state storage device is a high-speed secondary memory available on Cray PVP systems (except CRAY J90 series).

You can configure the SSD as a disk device used for filesystems or as a secondary data storage (SDS) device. See `ssdd`(4), `ssdt`(4), and `sds`(4). SDS space can be used for extended storage or can be configured as logical device cache with the `ldcache`(8) command. For more information, see `ldcache`(8).

When configured as a disk, the SSD functions as a fast random-access device that can be used for mounting file systems or for swapping. In this case, the SSD is represented in `/dev/ssdd` by one or more files. For more information about this configuration of the SSD, see `ssdd`(4) for IOS model E based systems or `ssdt`(4) for GigaRing-based systems.

FILES

`/dev/dsk`

`/dev/sds`

`/dev/ssdd`

`/dev/ssdt`

SEE ALSO

`sds`(4), `ssdd`(4), `ssdt`(4)

`sdss`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`ssbreak`(2), `ssread`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`ldcache`(8), `ldsync`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`ssdd` – SSD disk driver

IMPLEMENTATION

Cray PVP systems with an IOS model E

DESCRIPTION

The files in `/dev/ssdd` are character special files that allow read and write operations to SSD solid-state disk slices. Each file represents a one slice of the total SSD disk area. The total amount of the SSD allocated for SSD disks is specified in the UNICOS parameter file.

Usually, I/O request lengths to SSD disks must be in 512-word multiples and start on 512-word boundaries. A SSD disk slice can assume the attributes of a physical disk device. If the sector size of the specified device consists of more than 512 words, the I/O request lengths and starts must match those for the specified device.

Usually, you cannot mount the files in `/dev/ssdd` as file systems. You may specify a SSD disk slice as a whole or part of a logical disk device. You also can combine a SSD disk slice with a physical disk device. See `dsk`(4), `ldesc`(5), and `pdd`(4).

The files in `/dev/ssdd` are created by using the `mknod` command (see `mknod`(8)). Each must have a unique minor device number, a starting block, and a length (in blocks).

The `mknod`(8) command for SSD disk devices is as follows:

`mknod` *name type major minor dtype* `0` *start length*

name	Descriptive file name for the device.
type	Type of the device data being transferred. Devices in `/dev/ssdd` are character devices denoted by a `c`.
major	Major device number for SSD disk devices. The `dev_ssdd` name label in the `/uts/c1/cf/devsw.c` file denotes the major device number for SSD disk devices.
minor	Minor device number for this slice. Each SSD disk slice must have a unique minor device number.
dtype	Physical disk device type. This is optional. If left at 0, the SSD disk slice assumes the physical attributes of a DD-49 disk drive. For a list of physical disk device types, see `pdd`(4).
`0`	Placeholder for future use.
start	Absolute starting block (sector) number of the slice.
length	Number of blocks (sectors) in the slice.

FILES

```
/dev/ssdd/*
/usr/src/c1/io/ssdd.c
```

SEE ALSO

`dsk`(4), `ldd`(4), `ldesc`(5), `pdd`(4)

`mknod`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`ssdt` – GigaRing-based Solid State Disk storage device interface

IMPLEMENTATION

CRAY T90 systems

DESCRIPTION

The files in `/dev/ssdt` are special files that allow read and write operations to the GigaRing-based Solid State Disk storage device known as the SSD-T90. Each file represents one slice of an SSD-T90. The files in `/dev/ssdt` are character special files that may be used directly to read and write physical SSD-T90 slices.

You can configure the SSD-T90 as a disk device or as a secondary data storage (SDS) device.

When configured as a disk, the SSD-T90 functions as a fast random-access device that can be used for mounting file systems or for swapping. In this case, the SSD-T90 is represented in `/dev/ssdt` by one or more files. Usually, they are called to perform I/O on behalf of higher-level logical disk device drivers.

An `iounit` is a multiple of 4096-byte blocks that corresponds to the smallest read or write possible to a character special disk device. The `iounit` for an SSD-T90 device is normally 1, meaning read and write operations must transfer multiples of 4096 bytes and all seek operations must be on 4096-byte boundaries. The `iounit` may be set at a value greater than 1 as described below in the `mknod` command detail.

The files in `/dev/ssdt` are not usually mountable as file systems, although you may combine one or more physical disk slices to make a mountable logical disk device (see `dsk`(4), `ldd`(4), and `mount`(8)).

When configured as Secondary Data Storage (SDS), the SSD-T90 is managed in much the same way as main memory. It can be accessed directly by users with the `ssbreak`(2), `ssread`(2), and `sswrite`(2) system calls, or allocated as logical device cache for the caching of filesystem data. See `ssbreak`(2), `ssread`(2), `sswrite`(2), and `ldcache`(8).

The files in `/dev/ssdt` are created by using the `mknod`(8) command (see `mknod`(8)). Each must have a unique minor device number, along with other parameters used to define a physical disk slice.

The `mknod`(8) command for physical disk devices is as follows:

`mknod` *name type major minor dtype iopath start length flags reserved unit*

name	Descriptive file name for the device (for example, `/dev/ssdt/ssdt_blk0`).
type	Type of the device data being transferred. Devices in `/dev/ssdt` are character devices denoted by a `c`.
major	Major device number for physical disk devices. The `dev_ssdt` name label in the `/usr/src/uts/c1/cf/devsw.c` file denotes the major device number for physical disk devices. You can specify the major number as `dev_ssdt`.
minor	Minor device number for this slice.

dtype The *dtype* field is a compound field containing the `iounit` and target memory type for the SSD-T90. Target memory types are defined in the include file `sys/tmio.h`. The *dtype* field is broken down in octal as follows:

0*tttiiii* where:

ttt	= the target memory type
iiii	= the `iounit`

The SSD-T90 is either an 8 or 16 processor CRAY T3E. The value of the *dtype* field should be 0100001 for an 8 processor or 0110001 for a 16 processor T3E.

iopath The *iopath* specifies the GigaRing ring and node number that the SSD-T90 is connected to. It contains the following fields when broken down in octal:

0*rrrnn*0 where:

rrr	= GigaRing ring number
nn	= GigaRing node number

start Absolute starting block (`iounit` multiple) number of the slice.

length Number of blocks (`iounit` multiple) in the slice.

flags Flags for physical disk device control. They are mainly used for diagnostic and maintenance purposes. Usually the *flags* field should be 0 for slices in */dev/ssdt*.

reserved Currently unused. Should be 0.

unit Designates the SSD-T90 unit number. If only one SSD existis on a system, the unit should be 0.

Further information about configuring SSD-T90 for use as SDS memory can be found in *UNICOS Configuration Administrator's Guide*, publication SG-2303.

FILES

`/dev/ssdt/*`

`/usr/include/sys/ssdt.h`

`/usr/src/c1/io/ssdt.c`

SEE ALSO

`dsk`(4), `ldd`(4), `mdd`(4), `qdd`(4), `sdd`(4), `ssdd`(4),

`ddstat`(8), `mknod`(8), `mount`(8), `sdconf`(8), `sdstat`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`tape` – Physical tape device interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

Each tape device node in the `/dev/tape` directory provides an interface to a real physical tape device.

These devices attach to Cray Research systems with I/O subsystems model E using IBM Block Mux channels, ANSI intelligent peripheral interface (IPI) channels, Enterprise Systems Connection (ESCON) channels, or Small Computer System Interface (SCSI) channels. They attach to Cray Research GigaRing based systems using IBM Block Mux channels or SCSI channels.

The physical tape driver interface translates requests from a user or the tape daemon into requests packets that are sent to the attached I/O subsystem. The type determines the type of physical devices that may be attached. It manipulates the physical interface to accomplish the requested function and returns status to the system.

During system start-up, a file describing the tape configuration (`/etc/config/text_tapeconfig`) is read, tape device nodes are created in the `/dev/tape` directory, the configuration is sent to the tape driver and related I/O processors (IOPs), and the channels and control units are configured to the state specified in the tape configuration file. The permissions on these device paths are generally reserved for the root account and will have appropriate security labels.

You have two interfaces available for accessing tape devices:

Interface	**Description**
Character-special tape	Provides unstructured access to tape devices. Its capabilities provide tape access similar to the access that users on other UNIX systems have. This access is a basis means of reading and writing tape information.
Tape daemon-assisted (`tpddem`(4))	Intercepts user system call requests and processes requests from tape-related commands to perform tape resource management, device management, volume mounts and dismounts through operator communications or autoloader requests, label processing, volume switching, and error recovery. This interface is called the Tape Management Facility. The character-special tape interface does not provide these capabilities.

If the tape daemon-assisted interface is needed, executing the `tpdaemon`(8) command creates the daemon process that provides this interface. The tape daemon-assisted interface can use the configuration established during system start-up, or it can redefine the configuration. It operates concurrently with the character-special tape interface.

FILES

`/dev/tape/`*device_name*	Tape device node
`/etc/config/text_tapeconfig`	Tape subsystem configuration file

SEE ALSO

`tpddem`(4), `text_tapeconfig`(5)

Tape Subsystem User's Guide, Cray Research publication SG–2051

Tape Subsystem Administration, Cray Research publication SG–2307

NAME

`termio`, `termios` – General terminal interface

SYNOPSIS

```
#include <termio.h>
ioctl (int fildes, int request, struct termio *arg);
ioctl (int fildes, int request, int arg);

#include <termios.h>
ioctl (int fildes, int request, struct termios *arg);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

All of the asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this entry discusses the common features of this interface.

A terminal is associated with a terminal file in `/dev`. Terminal file names have the following form:

```
/dev/tty*
```

The user interface to this functionality is through function calls (the preferred interface) described on `terminal`(3C) man page or by the `ioctl`(2) requests described in this entry. This entry also discusses the common features of the terminal subsystem that are relevant to both user interfaces.

When a terminal file is opened, it usually causes the process to wait until a connection is established. In practice, a user's programs seldom open terminal files; they are opened by `getty`(8) and become a user's standard input, standard output, and standard error files. The very first terminal file opened by the session leader that is not already associated with a session becomes the controlling terminal for that session. The controlling terminal plays a special role in handling quit and interrupt signals; that role is discussed in this entry. The controlling terminal is inherited by a child process during a `fork`(2) system call. A process can break this association by changing its session using the `setsid`(2) system call.

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are lost only when the character input buffers of the system become completely full (for example, if the user has accumulated `MAX_INPUT` number of input characters that have not yet been read by some program); this situation is rare. (When the input limit is reached, all of the characters saved in the buffer up to that point are deleted without notice.)

Session Management (Job Control)

When a session is associated with a terminal, the control terminal designates one of the process groups as the foreground process group; all other process groups in the session are designated as background process groups. The foreground process group plays a special role in handling signal-generating input characters, as discussed in this entry. By default, when a controlling terminal is allocated, the controlling process's process group is assigned as foreground process group.

Background process groups in the controlling process's session are subject to a job control line discipline when they try to access their controlling terminal. Process groups can be sent signals that will cause them to stop, unless they have made other arrangements. An exception is made for members of orphaned process groups. These are process groups that do not have a member with a parent in another process group that is in the same session and therefore shares the same controlling terminal. When a member's orphaned process group tries to access its controlling terminal, errors will be returned because there is no process to continue it if it should stop.

If a member of a background process group tries to read its controlling terminal, its process group sent a `SIGTTIN` signal, which usually causes the members of that process group to stop. If, however, the process is ignoring or holding `SIGTTIN`, or is a member of an orphaned process group, the read operation will fail with an `errno` value set to `EIO`, and no signal will be sent.

If a member of a background process group tries to write its controlling terminal and the `TOSTOP` bit is set in the `c_lflag` field, its process group will be sent a `SIGTTOU` signal, which usually causes the members of that process group to stop. If, however, the process is ignoring or holding `SIGTTOU`, the write operation will succeed. If the process is not ignoring or holding `SIGTTOU` and is a member of an orphaned process group, the write operation will fail with `errno` set to `EIO`, and no signal will be sent.

If `TOSTOP` is set and a member of a background process group tries to issue an `ioctl`(2) system call to its controlling terminal, and that `ioctl` will modify terminal parameters (for example, `TCSETA`, `TCSETAW`, `TCSETAF`, or `TIOCSPGRP`), its process group will be sent a `SIGTTOU` signal, which usually causes the members of that process group to stop. If, however, the process is ignoring or holding `SIGTTOU`, the `ioctl`(2) will succeed. If the process is not ignoring or holding `SIGTTOU` and is a member of an orphaned process group, the write will fail with `errno` set to `EIO`, and no signal will be sent.

Canonical Mode Input Processing

Typically, all line editing and echoing functions are performed by the Cray Research system. You can off load this processing to the front-end system when using `telnet`(1B); however, not all of the features described in this entry are available in this mode.

Usually, terminal input is processed in units of lines. A line is delimited by a newline character (ASCII LF), an end-of-file character (ASCII EOT), or an end-of-line character. This means that a program trying a read operation is suspended until an entire line has been typed. Also, no matter how many characters are requested in the read operation, at most one line is returned. However, a whole line does not have to read at once; any number of characters may be requested in a read operation without loss of information.

Erase and kill processing is usually done during input. The `ERASE` character (by default, `#`) erases the last character typed. The `WERASE` character (`CONTROL-w`) erases the last *word* typed in the current input line (but not any preceding spaces or tabs). A *word* is defined as a sequence of nonblank characters, with tabs counted as blanks. Neither `ERASE` nor `WERASE` erases beyond the beginning of the line. The `KILL` character (by default, `@`) kills (deletes) the entire input line, and optionally outputs a newline character. All of these characters operate on a keystroke basis, independently of any backspaces or tabs that may have been entered. The `REPRINT` character (`CONTROL-r`) prints a newline character, followed by all characters that have not been read. Reprinting also occurs automatically if characters that usually would be erased from the screen are garbled by program output. The characters are reprinted as if they were being echoed; consequencely, if `ECHO` is not set, they are not printed.

You may enter both the erase and kill characters literally if they are preceded by the escape `\` symbol. In this case, the escape character is not read. You may change the erase and kill characters.

Noncanonical Mode Input Processing

In noncanonical mode input processing, input characters are not assembled into lines. Erase and kill processing does not occur. The `MIN` and `TIME` values are used to determine how to process the characters received, as follows:

`MIN`	Minimum number of characters that should be received when the read is satisfied (that is, when the characters are returned to the user).
`TIME`	Timer of 0.10-second granularity used to time-out transmissions that occur in bursts and short-term data transmissions.

The four possible combinations for `MIN` and `TIME` and their interactions are as follows:

Case A: `MIN` > 0, `TIME` > 0

In this case, `TIME` serves as an intercharacter timer and is activated after the first character is received. Because it is an intercharacter timer, it is reset after a character is received. The interaction between `MIN` and `TIME` is as follows. As soon as one character is received, the intercharacter timer is started. If `MIN` characters are received before the intercharacter timer expires (the timer is reset on receipt of each character), the read is satisfied. If the timer expires before `MIN` characters are received, the characters received to that point are returned to the user. If `TIME` expires, at least one character will be returned, because the timer would not have been enabled unless a character was received. In this case (`MIN` > 0, `TIME` > 0), the read sleeps until the `MIN` and `TIME` mechanisms are activated by the receipt of the first character. If the number of characters read is fewer than the number of characters available, the timer is not reactivated, and the subsequent read is satisfied immediately.

Case B: `MIN` > 0, `TIME` = 0

In this case, because the value of `TIME` is 0, the timer plays no role and only `MIN` is significant. A pending read is not satisfied until `MIN` characters are received (the pending read sleeps until `MIN` characters are received). A program that uses this case to read record-based terminal I/O may be blocked indefinitely in the read operation.

Case C: `MIN` = 0, `TIME` > 0

In this case, because `MIN` = 0, `TIME` no longer represents an intercharacter timer; it now serves as a read timer that is activated as soon as one `read` operation is requested. A read request is satisfied as soon as one character is received or the read timer expires. In this case, if the timer expires, no character is returned. If the timer does not expire, the read can be satisfied only if a character is received. In this case, the read will not block indefinitely waiting for a character; if no character is received within `TIME*`.10 seconds after the read is initiated, the read returns with zero characters.

Case D: `MIN` = 0, `TIME` = 0

In this case, return is immediate. The minimum of either the number of characters requested or the number of characters currently available is returned without waiting for more characters to be input.

The remainder of this subsection compares the different cases of interaction between the `MIN` and `TIME` values. In the following explanations, the interactions of `MIN` and `TIME` are not symmetric. For example, when `MIN` is greater than 0 and `TIME` equals 0, `TIME` has no effect. However, in the opposite case, in which `MIN` equals 0 and `TIME` is greater than 0, both `MIN` and `TIME` play a role in that `MIN` is satisfied with the receipt of one character. In case A (`MIN` greater than 0, `TIME` greater than 0), `TIME` represents an intercharacter timer; in case C (`TIME` equals 0, `TIME` greater than 0), `TIME` represents a read timer.

These two points highlight the dual purpose of the `MIN/TIME` feature. Cases A and B, in which `MIN` > 0, exist to handle burst mode activity (for example, file transfer programs), where a program would like to process at least `MIN` characters at a time. In case A, the intercharacter timer is activated by a user as a safety measure; in case B, the timer is turned off.

Cases C and D exist to handle single character, timed transfers. These cases are readily adaptable to screen-based applications that must know whether a character is present in the input queue before refreshing the screen. In case C, the read is timed; in case D, it is not.

Another important note is that `MIN` is always just a minimum. It does not denote a record length (for example, if a program does a read of 20 bytes, `MIN` is 10, and 25 characters are present, 20 characters will be returned to the user). When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. If echoing has been enabled, input characters are echoed as they are typed. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue is drained down to some threshold, the program is resumed. Certain characters have special functions on input. These functions and their default character values are summarized as follows:

`DISCARD`	(`CONTROL-o` or ASCII SI) describes subsequent output. Output is discarded until you type another `DISCARD` character, more input arrives, or a program you type clears the condition.
`DSUSP`	(`CONTROL-y` or ASCII EM) generates a suspend (`SIGTSTP`) signal, such as `SUSP`, but the signal is sent when a process in the foreground process group tries to read the `DSUSP` character, rather than when the character is typed.

`EOF`	(`CONTROL-d` or ASCII EOT) generates an end-of-file from a terminal. When this character is received, all of the characters that are waiting to be read are passed immediately to the program, without waiting for a newline character, and the `EOF` is discarded. If no characters are waiting (that is, the `EOF` occurred at the beginning of a line), 0 characters, the standard end-of-file indication, is passed back. The `EOF` character is not echoed unless it is escaped. Because ASCII EOT is the default `EOF` character, this prevents terminals that respond to EOT from hanging up.
`EOL`	(ASCII NUL) is an additional line delimiter, such as `NL`. Usually, it is not used.
`EOL2`	An additional line delimiter, such as `NL`. Usually, it is not used.
`ERASE`	(RUBOUT or ASCII DEL) erases the preceding character. It does not erase beyond the start of a line, as delimited by a `NL`, `EOF`, `EOL`, or `EOL2` character.
`INTR`	(`CONTROL-c` or ASCII ETX) generates an interrupt (`SIGINT`) signal that is sent to all frequent processes associated with the controlling terminal. Usually, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see `signal`(2).
`KILL`	(`CONTROL-u` or ASCII NAK) deletes the entire line, as delimited by a `NL`, `EOF`, `EOL`, or `EOL2` character.
`LNEXT`	(`CONTROL-v` or ASCII SYN) ignores the special meaning of the next character. This works for all of the special characters mentioned in this list. It allows characters to be input that would otherwise be interpreted by the system (for example, `KILL` or `QUIT`).
`NL`	(ASCII LF) is the normal line delimiter, but you can escape it by using the `LNEXT` character.
`QUIT`	(`CONTROL-|` or ASCII FS) generates a quit (`SIGQUIT`) signal. Its treatment is identical to the interrupt (`SIGINT`) signal, except that unless a receiving process has made other arrangements, it is terminated, and a core image file (called `core`) is created in the current working directory.
`REPRINT`	(`CONTROL-r` or ASCII DC2) reprints all characters, preceded by a newline character, that have not been read.
`START`	(`CONTROL-q` or ASCII DC1) resumes output that has been suspended by a `STOP` character. While output is not suspended, `START` characters are ignored.
`STOP`	(`CONTROL-s` or ASCII DC3) suspends output temporarily. It is useful with terminals to prevent output from disappearing before it can be read. While output is suspended, `STOP` characters are ignored.
`SUSP`	(`CONTROL-z` or ASCII SUB) generates a suspend (`SIGTSTP`) signal, which stops all processes in the foreground process group for that terminal.
`SWTCH`	(ASCII NUL) is reserved for future use.

WERASE	(CONTROL-w or ASCII ETX) erases the preceding word. (A *word* is defined as a sequence of nonblank characters, with tabs counted as blanks.) It does not erase beyond the start of a line, as delimited by a NL, EOF, EOL, or EOL2 character.

You may change all character values except NL to suit individual tastes. If the value of a special control character is _POSIX_VDISABLE (0), the function of that special control character is disabled. To escape the ERASE, KILL, and EOF characters, use a preceding \ symbol; in which case, no special function is performed. You can precede any of the special characters by the LNEXT character; in which case, no special function is performed.

Modem Disconnect

When a modem disconnect is detected, a hang-up (SIGHUP) signal is sent to the terminal's controlling process. Unless other arrangements have been made, these signals terminate the process. If SIGHUP is ignored or caught, any subsequent read operation returns with an end-of-file (EOF) indication until the terminal is closed.

If the controlling process is not in the foreground process group of the terminal, a SIGTSTP is sent to the terminal's foreground process group. Unless other arrangements have been made, these signals stop the processes.

Processes in background process groups that try to access the controlling terminal after modem disconnect while the terminal is still allocated to the session will receive appropriate SIGTTOU and SIGTTIN signals. Unless other arrangements have been made, this signal stops the processes.

The controlling terminal remains in this state until it is reinitialized with a successful open by the controlling process, or deallocated by the controlling process. The parameters that control the behavior of devices and modules providing the termios interface are specified by the termios structure defined by the sys/termios.h include file. Several ioctl(2) requests apply to terminal files. The primary requests use the following structure, defined in the sys/termios.h include file:

```
tcflag_t     c_iflag;           /* input modes   */
tcflag_t     c_oflag;           /* output modes  */
tcflag_t     c_cflag;           /* control modes */
tcflag_t     c_lflag;           /* local modes   */
cc_t         c_cc[NCCS];        /* control characters */
```

The c_cc array defines the special control characters. The symbolic name NCCS is the size of the control-character array and also is defined by termios.h. The relative positions, subscript names, and typical default values for each function are as follows:

0	VINTR	DEL
1	VQUIT	FS
2	VERSE	#
3	VKILL	@
4	VEOF	EOT
5	VEOL	NUL
6	VEOL2	NUL
7	VSWTCH	NUL
8	VSTRT	DC1
9	VSTOP	DC3
10	VSUSP	SUB
11	VDSUSP	EM
12	VREPRINT	DC2
13	VDISCRD	SI
14	VWERSE	ETB
15	VLNEXT	SYN
16-19	Reserved	

Input Modes

The `c_iflag` field describes the basic terminal input control, as follows:

IGNBRK	Ignores break condition
BRKINT	Signals interrupt on break
IGNPAR	Ignores characters with parity errors
PARMRK	Marks parity errors
INPCK	Enables input parity check
ISTRIP	Strips character
INLCR	Maps `NL` to `CR` on input
IGNCR	Ignores CR
ICRNL	Maps `CR` to `NL` on input
IUCLC	Maps uppercase to lowercase on input
IXON	Enables start and stop output control
IXANY	Enables any character to restart output
IXOFF	Enables start and stop input control

The initial input control value is `BRKINT`, `IGNPAR`, `ISTRIP` `IXON`, and `IXANY`. If set, the bits have the following meanings:

`IGNBRK`	Ignores the break condition (a character-framing error with data that consists of all 0's); that is, nothing is put on the input queue and therefore, a process does not read any break character. Otherwise, if `BRKINT` is set, the break condition flushes the input and output queues and, if the terminal is the controlling terminal of a foreground process group, sends the interrupt (`SIGINT`) signal to that foreground process group. Otherwise, if neither `IGNBRK` nor `BRKINT` is set, a break condition is read as a single ASCII NULL character (´\0´) (if `PARMRK` is set, it is read as as ´\377´, ´\0´, ´\0´).
`BRKINT`	Generates an interrupt signal for the break condition and flushes both the input and output queues if `IGNBRK` is set.
`IGNPAR`	Ignores bytes that have framing or parity errors (other than break).
`PARMRK`	Reads any character that has a framing or parity error, other than break, that is not ignored (`IGNPAR` is not set) as the 3-character sequence 0377, 0, *X*; *X* is the data of the character received in error. To avoid ambiguity in this case, a valid character of 0377 is read as 0377, 0377 if `ISTRIP` is not set. If neither `IGNPAR` nor `PARMRK` is set, a character that has a framing or parity error (other than break) is read as a single ASCII NULL character (´\0´).
`INPCK`	Enables input parity checking. If `INPCK` is not set, input parity checking is disabled. This allows output parity generation without input parity errors. Whether input parity checking is enabled or disabled is independent of whether parity detection is enabled or disabled. If parity detection is enabled but input parity checking is disabled, the hardware to which the terminal is connected will recognize the parity bit, but the terminal special file will not check whether this is set correctly.
`ISTRIP`	Strips valid input characters to 7 bits; if `ISTRIP` is not set, all 8 bits are processed.
`INLCR`	Translates a received `NL` character into a `CR` character. If `IGNCR` is set, a received `CR` character is ignored; otherwise, if `ICRNL` is set, a received `CR` character is translated into a `NL` character.
`IGNCR`	Ignores a received `CR` character. If `IGNCR` is not set, and `ICRNL` is set, a received `CR` character is translated into an `NL` character.
`ICRNL`	Translates a received `CR` character into an `NL` character if `IGNCR` is not set.
`IUCLC`	Translates a received uppercase alphabetic character into the corresponding lowercase character.
`IXON`	Enables start and stop output control; a received `STOP` character suspends output, and a received `START` character restarts output. The `STOP` and `START` characters are not read, but they merely perform flow control functions. If `IXANY` is set, any input character restarts output that has been suspended.
`IXANY`	Any input character restarts suspended output.
`IXOFF`	Transmits a `START` character when the input queue is nearly empty and a `STOP` character when the input queue is nearly full.

Output Modes

The `c_oflag` field specifies the system treatment of output, as follows:

`OPOST`	Postprocesses output
`OLCUC`	Maps lowercase to uppercase on output
`ONLCR`	Maps `NL` to `CR-NL` on output
`OCRNL`	Maps `CR` to `NL` on output
`ONOCR`	No `CR` output at column 0
`ONLRET`	`NL` performs `CR` function
`OFILL`	Uses fill characters for delay
`OFDEL`	Fill is `DEL`, else `NULL`

`NLDLY`	Selects newline delays:
	`NL0`
	`NL1`

`CRDLY`	Selects carriage-return delays:
	`CR0`
	`CR1`
	`CR2`
	`CR3`

`TABDLY`	Selects horizontal tab delays or tab expansion:
	`TAB0`
	`TAB1`
	`TAB2`

`TAB3`	Expands tabs to spaces.

`BSDLY`	Selects backspace delays:
	`BS0`
	`BS1`

`VTDLY`	Selects vertical tab delays:
	`VT0`
	`VT1`

`FFDLY`	Selects form feed delays:
	`FF0`
	`FF1`

The initial output control value is `OPOST`, `ONLCR`, and `TAB3`. If set, the bits have the following meanings:

`OPOST`	Postprocesses output characters as indicated by the remaining flags; if `OPOST` is not set, characters are transmitted without change.
`OLCUC`	Transmits a lowercase alphabetic character as the corresponding uppercase character. This function is often used in conjunction with `IUCLC`.
`ONLCR`	Transmits the `NL` character as the `CR-NL` character pair.
`OCRNL`	Transmits the `CR` character as the `NL` character.
`ONOCR`	Does not transmit a `CR` character when at column 0 (first position).
`ONLRET`	Performs the `CR` function. The `NL` character is assumed to do the carriage-return function; the column pointer is set to 0, and the delays specified for `CR` are used. Otherwise, the `NL` character is assumed to do just the line-feed function; the column pointer remains unchanged. If the `CR` character is actually transmitted, the column pointer also is set to 0.

The following delay bits specify the length of time the transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases, a value of 0 indicates no delay.

`OFILL`	Transmits fill characters for a delay instead of a timed delay. This is useful for terminals that have a high baud rate that need only a minimal delay.
`OFDEL`	Sets the fill character to `DEL` (`NULL` by default).
`NLDLY`	Selects newline delays. Newline delay lasts for about 0.10 seconds. If `ONLRET` is set, the carriage-return delays are used instead of the newline delays. If `OFILL` is set, two fill characters are transmitted.
`CRDLY`	Selects carriage-return delays. Carriage-return delay type 1 depends on the current column position; type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If `OFILL` is set, delay type 1 transmits two fill characters, and type 2 transmits four fill characters.
`TABDLY`	Selects horizontal tab delays. Horizontal-tab delay type 1 depends on the current column position; type 2 is about 0.10 seconds, and type 3 specifies that tabs will be expanded into spaces. If `OFILL` is set, two fill characters are transmitted for any delay.
`TAB3`	Expands tabs to spaces.
`BSDLY`	Selects backspace delays. Backspace delay lasts for about 0.05 seconds. If `OFILL` is set, one fill character is transmitted.
`VTDLY`	Selects vertical tab delays. Vertical-tab delay lasts for about 2 seconds. If `OFILL` is set, two fill characters are transmitted.
`FFDLY`	Selects form-feed delays. Form-feed delay lasts for about 2 seconds. If `OFILL` is set, two fill characters are transmitted.

The actual delays depend on line speed and system load.

Control Modes

The `c_cflag` field describes the hardware control of the terminal, as follows:

`CBAUD`	Baud rate:	
	`B0`	Hang up
	`B50`	50 Bd
	`B75`	75 Bd
	`B110`	110 Bd
	`B134`	134 Bd
	`B150`	150 Bd
	`B200`	200 Bd
	`B300`	300 Bd
	`B600`	600 Bd
	`B1200`	1200 Bd
	`B1800`	1800 Bd
	`B2400`	2400 Bd
	`B4800`	4800 Bd
	`B9600`	9600 Bd
	`B19200`	19200 Bd
	`EXTA`	External A
	`B38400`	38400 Bd `EXTB` External B
`CSIZE`	Character size:	
	`CS5`	5 bits
	`CS6`	6 bits
	`CS7`	7 bits
	`CS8`	8 bits
`CSTOPB`	Sends 2 stop bits, else 1	
`CREAD`	Enables receiver	
`PARENB`	Enables parity	
`PARODD`	Odd parity, else even	
`HUPCL`	Hangs up on last close	

`CLOCAL`	Local line, else dial-up

The initial hardware control value after an `open`(2) system call is `B9600`, `CS8`, `CREAD`, and `HUPCL`.

If set, the bits in the `c_cflag` field have the following meanings:

`CBAUD`	Specifies the baud rate. The 0 Bd rate, `B0`, hangs up the connection. If `B0` is specified, the data-terminal-ready signal is not asserted. Usually, this disconnects the line. For any particular hardware, impossible speed changes are ignored. The default is `B9600`, which specifies a 9600 Bd rate.
`CSIZE`	Specifies the character size (in bits) for both transmission and reception. This size does not include the parity bit if any. The default is `CS8`, which specifies a character size of 8 bits.
`CSTOPB`	Specifies the number of stop bits used. If `CSTOPB` is set, 2 stop bits are used; otherwise, 1 stop bit is used (for example, at 110 Bd, 2 stops bits are required).
`CREAD`	Enables the receiver. If `CREAD` is not set, no characters are received. `CREAD` is set by default.
`PARENB`	Enables parity generation and detection, and adds a parity bit to each character. If parity is enabled and the `PARODD` flag is set, odd parity is used; otherwise, even parity is used.
`PARODD`	Specifies odd parity if `PARENB` is set.
`HUPCL`	Disconnects the line when the last process with that line open closes it or terminates; that is, the data-terminal-ready signal is not asserted. `HUPCL` is set by default.
`CLOCAL`	If `CLOCAL` is set, the line is assumed to be a local, direct connection with no modem control. If `CLOCAL` is not set, modem control is assumed.

Local Modes

The `c_lflag` field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline provides the following:

`ISIG`	Enables signals
`ICANON`	Enables canonical input (erase and kill processing)
`XCASE`	Enables canonical upper/lower presentation
`ECHO`	Enables echo
`ECHOE`	Echoes erase character as `BS-SP-BS`
`ECHOK`	Echoes `NL` after kill character
`ECHONL`	Echoes `NL`
`NOFLSH`	Disables flush after interrupt or quit
`TOSTOP`	Sends `SIGTTOU` for background output
`IEXTEN`	Enables extended (implementation-defined) functions

The initial line-discipline control value is ISIG, ICANON, ECHO, and ECHOK.

If set, the bits have the following meanings:

ISIG Enables signals. Each input character is checked against the special control characters INTR, QUIT, SWTCH, SUSP, STATUS, and DSUSP. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, checking is not done. Thus, these special input functions are possible only when ISIG is set. To disable these functions individually, change the value of the control character to an unlikely or impossible value (for example, 0377).

ICANON Enables canonical processing. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, EOL, and EOL2. If ICANON is not set, read requests are satisfied directly from the input queue. A read is not satisfied until at least MIN characters have been received, or the time-out value TIME has expired between characters. This allows fast bursts of input to be read efficiently while still allowing single-character input. The MIN and TIME values are stored in the position for the EOF and EOL characters, respectively.

The time value is represented in tenths of seconds.

XCASE Specifies canonical presentation of uppercase and lowercase characters. If XCASE is set and ICANON also is set, an uppercase letter is accepted on input when prefaced by a \ character, and an uppercase letter is prefaced by a \ character on output. In this mode, the following escape sequences are generated on output and accepted on input:

For:	Use:
`	\´
|	\!
~	\^
{	\(
}	\)
\	\\

For example, A is input as \a, \n as \\n, and \N as \\\n.

ECHO Enables echo. If ECHO is set, characters are echoed as received.

ECHOE Echoes the erase character as ASCII BS SP BS.

ECHOK Echoes an NL character after the kill character.

ECHONL Echoes an NL character.

NOFLSH Disables the normal flush of the input and output queues associated with the interrupt (INTR), quit (QUIT), and suspend (SUSP) characters. This bit should be set when restarting system calls that read from or write to a terminal; see sigaction(2).

TOSTOP	Sends the SIGTTOU signal for background output. If a process tries to write to its controlling terminal when it is not in the foreground process group for that terminal, the signal is sent. This signal usually stops the process. Otherwise, the output generated by that process is output to the current output stream. Processes that are blocking or ignoring SIGTTOU signals are excepted and allowed to produce output, if any.
IEXTEN	Enables the following extended (implementation-defined) functions: special characters (WERASE, REPRINT, DISCARD, and LNEXT) and local flags (TOSTOP, ECHOCTL, ECHOPRT, ECHOKE, FLUSHO, and PENDIN).

When ICANON is set, the following echo functions are possible:

- If ECHO and ECHOE are set, the erase character (ERASE and WERASE) is echoed as ASCII BS SP BS, which clears the last character from a terminal.
- If ECHOK is set and ECHOKE is not set, the NL character is echoed after the kill character to emphasize that the line is deleted. An escape character (\) or an LNEXT character that precedes the erase or kill character removes any special function.
- If ECHONL is set, the NL character is echoed even if ECHO is not set. This is useful for terminals set to local echo (half-duplex).

Minimum and Time-out

The MIN and TIME values are described in the Noncanonical Mode Input Processing subsection. The initial value of MIN is 1, and the initial value of TIME is 0.

Terminal Size

The number of lines and columns on the terminal's display is specified in the winsize structure defined by sys/termios.h, which includes the following members:

```
unsigned  short   ws_row;     /* rows, in characters        */
unsigned  short   ws_col;     /* columns, in characters     */
unsigned  short   ws_xpixel;  /* horizontal size, in pixels */
unsigned  short   ws_ypixel;  /* vertical size, in pixels   */
```

termio Structure

Some ioctl(2) requests use the termio structure. It is defined by the sys/termio.h include file and includes the following members:

```
unsigned     short     c_iflag;     /* input modes         */
unsigned     short     c_oflag;     /* output modes        */
unsigned     short     c_cflag;     /* control modes       */
unsigned     short     c_lflag;     /* local modes         */
char                   c_line;      /* line discipline     */
unsigned     char      c_cc[NCC];   /* control characters */
```

The `c_cc` array defines the special control characters. The symbolic name `NCC` is the size of the control-character array and also is defined by the `sys/termio.h` include file. The relative positions, subscript names, and typical default values for each function are as follows:

0	VINTR	DEL
1	VQUIT	FS
2	VERASE	#
3	VKILL	@
4	VEOF	EOT
5	VEOL	NUL
6	VEOL2	NUL
7	reserved	

The calls that use the `termio` structure affect only the flags and control characters that can be stored in the `termio` structure; all other flags and control characters are unaffected.

Supported `ioctl` Requests

This subsection lists the primary `ioctl`(2) requests supported by devices and STREAMS modules providing the `termios` interface (see `terminal`(3C)). All devices or modules may not support some `ioctl`(2) requests. The functionality provided by these requests also is available through the preferred function call interface specified on the `terminal`(3C) man page.

The following `ioctl` requests are supported:

TCFLSH	Flushes input and/or output queues. If *arg* is 0, `TCFLSH` flushes the input queue; if *arg* is 1, it flushes the output queue; if *arg* is 2, it flushes both the input and output queues.
TCGETA	Gets the parameters associated with the terminal and stores them in the `termio` structure referenced by *arg*.
TCSBRK	Waits for the output to drain. If *arg* is 0, `TCSBRK` sends a break (0 bits for 0.25 seconds).
TCSETA	Sets the parameters associated with the terminal from the `termio` structure referenced by *arg*. The change is immediate.
TCSETAF	Waits for the output queue to empty, then flushes the input queue and sets the new parameters from the `termio` structure referenced by *arg*.
TCSETAW	Waits for the output queue to empty before setting the new parameters from the `termio` structure referenced by *arg*. When changing parameters that affect output, use this request.
TCXONC	Enables start and stop control. If *arg* is 0, `TCXONC` suspends output; if *arg* is 1, it restarts suspended output; if *arg* is 2, it suspends input; and if *arg* is 3, it restarts suspended input.
TIOCGWINSZ	Stores the terminal size in the `winsize` structure to which *arg* points.
TIOCSWINSZ	Sets the terminal size from the `winsize` structure to which *arg* points. If the new sizes are different from the old sizes, a `SIGWINCH` signal is set to the process group of the terminal.
TCCLRCTTY	Clears the controlling tty connection.

`TCSETCTTY`	Defines the terminal as the controlling tty for a session.
`TCGETPGRP`	Gets the foreground process group ID for the session.
`TCSETPGRP`	Sets the foreground process group ID for the session.
`TCSIG`	Interrupts all outstanding asynchronous I/O in the foreground process group, and it sends the process group a signal.
`TCGETDEV`	Gets the device number of the terminal.
`TCTTRD`	Reserved for use by the SCP Interactive facility.
`TCRDFL`	Enables the "master read failure with user reads" option.

FILES

`/dev/*`

`/dev/tty*`

`/usr/include/termio.h`

SEE ALSO

`stty`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`fork`(2), `ioctl`(2), `setpgrp`(2), `setsid`(2), `sigaction`(2), `signal`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`terminal`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`getty`(8), `telnetd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

tpddem – Tape daemon interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The tape pseudo device driver provides user-level control over the tape devices. That is, the tape daemon (TPD) interface. The tpddem interface provides a mechanism for the tape daemon to control the tape devices. The daemon handles allocation and scheduling of tape devices. Only a process with a sysadm category may have direct access to a TPD device.

The tpddem interface contains hooks at critical points such as open operations, the first I/O operation, error recovery, and close operations. The tape daemon uses these hooks to control user access to the tape devices, including validation of open operations, automatic volume switching, label processing, and error recovery. When the tape daemon activates one of these hooks, the tpddem interface suspends the current user process and sends a signal to the tape daemon. When the controlling process completes, it sends an ioctl(2) system call to the tpddem driver to resume the user process. The controlling process may resume the user process that has an error.

The hooks in the tape device drivers cause the tape daemon process that has the TPD device open to preempt the user process that is using a tape device. If no process has TPD open, the tape driver falls through these hooks.

The tpddem interface supports the close(2), ioctl(2), and open(2) system calls. The open(2) system call activates hooks in the TPD driver for the calling process. The super user must run the calling process; the device is exclusive (only one process may have the device open). The close(2) system call deactivates the hooks in the TPD driver. The ioctl(2) system call sends commands to the tape driver.

The tpddem structure that follows, as defined in the sys/tpddem.h include file, is used to communicate between the driver and the controlling process.

```
/* TPD daemon control table */

struct tpddem {
       int     flags;              /* open/close flags             */
       int     owner;              /* ID of controlling process    */
       int     pid;                /* process ID of tape demon     */
       struct  proc *p;            /* proc address of controlling  */
                                   /* process                      */
       long    demdev;             /* tpddev device (major, minor) */
       long    reserved0;          /* reserved for future use      */
       long    reserved1;          /* reserved for future use      */
       long    reserved2;          /* reserved for future use      */
       long    reserved3;          /* reserved for future use      */
       struct  bdtab tab[TPD_MAXBMXDV]; /* substructure 1 per device */
};

 /* TPD device control/status structure */

 struct bdtab {
        word  name;    /* Name of device                            */
        int   dev;     /* Minor device number                       */
        int   flag;    /* Device requires daemon process            */
        int   func;    /* Current tpd device function               */
        int   user;    /* User ID                                   */
        long  status;  /* Current device status                     */
        int   error;   /* Error return to user                      */
        int   reslct;  /* Device in reselect                        */
        int   wait;    /* User sleeping on this table               */
        int   rval;    /* Function-dependent return value           */

 /* Following used for reselect to new device */

        int  newdev;   /* Ordinal of new device table               */

 /* For tape positioning and user end-of-volume communication       */

        int   dmn_int1
        int   dmn_int2
        int   dmn_int3
        int   dmn_int4
        int   partition;      /* Partition to position to           */
        int   filesec;        /* File section to position to        */
        int   datablock;      /* Datablock to position to           */
        int   absaddr;        /* Absolute address to position to    */
```

```
               long  dmn_tpvdev;     /* tpd device number                  */
               long  oflags;         /* open flags                         */
               long  dmn_reserved1;  /* reserved for future use            */
               long  dmn_reserved2;  /* reserved for future use            */
               long  dmn_reserved3;  /* reserved for future use            */
          };
```

The available `ioctl` requests, as defined in the `sys/tpddem.h` include file, are as follows:

`TDM_BFMON`	Buffer monitor.
`TDM_CUEOV`	Clears user end-of-volume (EOV) flag.
`TDM_FIRST`	First daemon function.
`TDM_GET`	Returns the structure `tpddem` to the passed-in address.
`TDM_OBIT`	Returns a list of job IDs for jobs that have terminated and are still recorded in the system reservation table.
`TDM_RLS`	Removes the passed-in job ID from the system reservation table.
`TDM_RSL`	Reselects to a new device.
`TDM_RSM`	Resumes the user process for the device specified by the passed-in structure `bdtab`. The `error` member of `bdtab` is set into the user's error status. This request must follow the `TDM_SET` request.
`TDM_RSV`	Saves the process group ID passed in the system reservation table. When the last member of a process group exits and it is recorded in this table, a signal is sent to the controlling process for resource control.
`TDM_SET`	Sets the fields in the `bdtab` system structure from the passed-in structure `bdtab`. Also clears the `flag` member of `bdtab`. This request must precede the `TDM_RSM` request.
`TDM_SUEOV`	Sets user EOV flag.
`TDM_WDN`	Waits for the specified device to complete its current operation.

The tape daemon reason codes, which are used to signal the tape daemon, are as follows:

`TDR_ABN`	`abn` flag set.
`TDR_CLOSE`	User `close` flag.
`TDR_DEMPROC`	Tape daemon processing flag.
`TDR_IO`	First I/O request.
`TDR_OPEN`	User `open` flag.
`TDR_RDONLY`	Request to write on a read-only file.
`TDR_REASON`	First reason code flag. All reason codes must be greater than this value.

`TDR_USRREQ`	User function.
`TDM_WTM`	Write tape mark flag.

FILES

`/dev/bxmdem`	Tape daemon (TPD) interface
`/usr/include/sys/tpddem.h`	Structure definition of `tpddem`

SEE ALSO

`tape`(4)

`close`(2), `ioctl`(2), `open`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

Tape Subsystem User's Guide, Cray Research publication SG–2051

NAME

`tty` – Controlling terminal interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `/dev/tty` file is a synonym for the control terminal, if any, associated with the process group of each process. It is useful for programs or shell sequences that want to be sure of writing messages on the terminal no matter how output has been redirected. When output to the terminal is desired, you also can use `/dev/tty` for programs that require the name of a file for output. In this way, the program does not have to find out the terminal that is currently in use.

FILES

`/dev/tty`

SEE ALSO

`pty`(4), `termio`(4), `tp`(4) (CRAY Y-MP systems)

NAME

`vme` – VME (FEI-3) network interface

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The VME (FEI-3) network interface is a channel-to-VME backplane adapter that connects a Cray Research system with a VMEbus based, front-end computer.

The special files for the FEI-3 interface are in the `/dev` directory.

FEI-3 special file names have the following naming convention:

`/dev/vme`*nn*

nn Minor device number ("logical path" in IOS terminology) for the VME interface

Support for the FEI-3 is provided through the IOS as if the FEI-3 were an NSC adapter. The device is otherwise treated as an NSC adapter; for more information, see `hy`(4).

FILES

`/dev/vme*`

`/usr/include/sys/hy.h`

`/usr/include/sys/hysys.h`

SEE ALSO

`hy`(4)

`ioctl`(2), `listio`(2), `read`(2), `reada`(2), `write`(2), `writea`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

`xdd` – Physical disk device interface

IMPLEMENTATION

CRAY J90se systems

CRAY T90 systems

DESCRIPTION

The files in `/dev/xdd` are special files that allow read and write operations to physical disk devicesconnected to the MPN-1 (SCSI disks), the FCN-1 (Fibre SCSI disks), or the HPN-1/HPN-2 (HIPPI disks). Each file represents one slice of a physical disk device. The files in `/dev/xdd` are character special files that may be used directly to read and write physical disk slices. Usually, they are called to perform I/O on behalf of higher-level logical disk device drivers. For I/O on a character disk device, read and write operations must transfer multiples of the physical device sector size and all seek operations must be on physical sector size boundaries.

The files in `/dev/xdd` are not usually mountable as file systems, although you may combine one or more physical disk slices to make a mountable logical disk device (see `dsk`(4), `ldd`(4), and `mount`(8)).

The files in `/dev/xdd` are created by using the `mknod` command (see `mknod`(8)). Each must have a unique minor device number, along with other parameters used to define a physical disk slice.

The `mknod`(8) command for physical disk devices is as follows:

`mknod` *name type major minor dtype iopath start length flags altpath unit* [*ifield*]

name Descriptive file name for the device (for example, `xdd/scr0230`).

type Type of the device data being transferred. Devices in `/dev/xdd` are character devices denoted by a `c`.

major Major device number for physical disk devices. The `dev_xdd` name label in the `/usr/src/uts/c1/cf/devsw.c` file denotes the major device number for physical disk devices. You can specify the major number as `dev_xdd`.

minor Minor device number for this slice.

dtype Physical disk device type defition, consisting of 32 bits defined as follows:

Bit 0 - 11	Defines the sector size.
Bit 12 - 19	Defines the type field. Currently this field is not used.
Mode bit	Mode bit. If mode is 1, the sector size given is in words per sector. If mode is 0 (the default value), the sector size given is *iouni ts*. An *iounit* is 512 words. The mode bit is not currently supported.

iopath Specifies the GigaRing number on which the device is located, the node number to which the disk is connected, and the controller slot number (ION channel number) of the device.

Bit 0 - 2	Defines the controller slot number.
Bit 3 - 8	Defines the ION node number.
Bit 9 - 15	Defines the Ring number.

For example: An *iopath* of 0110204(octal) indicates Controller Number 4, ION node number 2, and Ring Number 011(octal) or 9(decimal).

For disk devices connected to the MPN-1 (SCSI disks), the SCSI controller slot number is in the range 0 through 8. For disk devices connected to the FCN-1 (Fibre SCSI disks), the controller number is in the range 0 through 4.

For HIPPI disks, the *iopath* is represented by an octal number in the format 0*rrrnnc* where:

rrr	The GigaRing number
nn	The node number of the HPN
c	The channel on the HPN

start Absolute starting sector number of the slice.

length Number of blocks (sectors) in the slice.

flags Flags for physical disk device control. They are mainly used for diagnostic and maintenance purposes. Usually, the flags field should be 0 for slices in `/dev/xdd`. For HIPPI disks, the *flags* field is 0.

```
#define S_CONTROL       001    /* control device                 */
#define S_NOBBF         002    /* no bad block forwarding        */
#define S_NOERREC       004    /* no error recovery              */
#define S_NOLOG         010    /* no error logging               */
#define S_NOWRITEB      020    /* no write behind                */
#define S_CWE           040    /* control device write enable    */
#define S_NOSORT        0200   /* no disk sort                   */
#define S_NODEVINT      0400   /* no device intimate functions   */
#define S_MODE_MASK     0777   /* Mask containing special flags  */
```

altpath Optional alternate *iopath* that you can use as a back-up path to the physical disk device’s second port.

unit The disk device unit number for device types that support multiple units on the same channel.

The unit number is defined as follows:

Bit 0 - 7	Disk unit number
Bit 8 - 15	Logical Unit Number

For HIPPI disks, this is the disk’s facility number.

ifield (HIPPI disks only). The hardware address of the HIPPI disk in the HIPPI network.

FILES

`/dev/xdd/*`

`/usr/include/sys/xdd.h`

`/usr/src/c1/io/xdd.c`

SEE ALSO

`dsk`(4), `ldd`(4), `mdd`(4), `qdd`(4), `sdd`(4),

`ddstat`(8), `mknod`(8), `mount`(8), `sdconf`(8), `sdstat`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022