NAME

intro – introduction to special files

DESCRIPTION

This section describes various special files that refer to specific hardware peripherals and operating system device drivers. The names of the entries are generally derived from names for the hardware, as opposed to the names of the special files themselves. Characteristics of both the hardware device and the corresponding device driver are discussed where applicable.

NAME

console – console terminal

DESCRIPTION

The special file **console** designates a standard destination for system diagnostics. The kernel writes its diagnostics to this file, as does any user process with messages of systemwide importance. If **console** is associated with a physical terminal, than console messages also appear on that terminal; but it is quite standard not to have **console** associated with a physical terminal.

Note that *inittab*(4) does not normally post a *getty* on **console.** This is because **console** might become associated with a terminal that already is a login terminal.

Console conventions differ between System 6300 and System 6600 Systems.

MiniFrame

Console messages are saved in a circular buffer. Reading **console** retrieves the messages and removes them from the buffer.

If the operating system is configured with the kernel debugger (see *config*(1M), then **tty000** is associated with the console. This means that console messages also go to **tty000** and that a Control-B on **tty000** starts the kernel debugger.

The size of the console circular buffer is configured with the *config*(1M) parameter **cbufsize**. The default is 512 bytes.

The following *ioctl*(2) commands are acceptd:

ioctl(fd, CONERR);
: *Fd* must be open to **console**. All console output is to be duplicated in the error message queue. See *err (7).*

ioctl(fd, CONBUF);
: *Fd* must be open to **console**. No console output is to be duplicated in the error message queue. This is the initial condition.

MegaFrame

Each Application Processor has its own console, which can be associated with any terminal or with no terminal at all. Whether or not the console is associated with a terminal, the most recent console output is saved in a circular buffer.

Input/output operations on **console** by a process running on an Application Processor affect the console for that Application Processor. The exact meaning depends on whether or not the console is associated with a terminal.

- If the console is associated with a terminal, all input/output operations to **console**, including *ioctl*(2), have the same effect as if applied directly to the terminal, except that output is duplicated in the console buffer.
- If the console isn't associated with a terminal, all attempts to read the console return an end of file condition, all writes to the console go only to the console buffer, and *ioctl* operations have no affect on any terminal.

If the kernel debugger is enabled, a Control-B or Code-B on the terminal associated with the console activates the kernel debugger. The command "go" to the kernel debugger resumes normal processing.

The *console*(1M) command and *syslocal*(2) system calls control terminal association and print the buffers of System 6600 Application Processor consoles.

FILES

/dev/console

SEE ALSO

conlocate(1M), console(1M), syslocal(2).

WARNING

The kernel debugger is not a supported product and may disappear without warning. Normal system processing is suspended while the kernel debugger is active.

NAME

err – error-logging interface

DESCRIPTION

Minor device 0 of the *err* driver is the interface between a process and the system's error-record collection routines. The driver may be opened only for reading by a single process with super-user permissions. Each read causes an entire error record to be retrieved and removed; the record is truncated if the read request is for less than the record's length.

An appropriate command to the console sends console information to the error record queue. See *console*(7).

FILES

/dev/error special file

SEE ALSO

errdemon(1M), console(7).

NAME

cstermio - terminal I/O character set interface

DESCRIPTION

Terminal input-character sequences can be translated from device-dependent character codes to internal character-set representations (XSIS 058404 character-code standard, including Motorola private character-set 040) prior to receipt by a user process. Similarly, outbound characters destined for a terminal can be translated from internal character-set representations to device-dependent output sequences. Such character-set translations are handled by the terminal driver.

There are two aspects to terminal I/O character-set handling:

- Managing terminal I/O translation tables at the system-wide level.
- Controlling terminal I/O character-set translation options for each active terminal.

Managing terminal I/O translation tables is a superuser responsibility and can be done through the itt(1M) command. An interface to translation table management is also provided through the ioctl(2) system call. Ioctl(2) accommodates status requests and changes to data and option settings.

Csttbl Structure:

The ioctl(2) system calls that apply to managing terminal I/O character-set translation tables use the structure shown in Figure Cstermio-1 and the structures pointed to therein. All of these structures are defined in <cstty.h>. The fields in this structure are explained in Table Cstermio-1.

```
/*
 * Character set translation table argument for CSGETTT and CSSETTT.
 * The user program should define CSMAXSIZ as the maximum translation
 * table size it is prepared to handle and set cs_tmax to that value.
 */
```

Figure Cstermio-1. Csttbl Structure (Page 1 of 2)

```
#ifdef CSMAXSIZ
struct  csttbl   {
        int          cs_tmax;          /* should be set to CSMAXSIZ */
        union    {
                 struct cstthdr  cs_hdr;
                 char            cs_tbl[CSMAXSIZ];
        }cs_u;
};
#endif

/* description of a character set translation table header */
struct cstthdr {
   ushort   cs_tnum;         /* table number */
   ushort   cs_tlen;         /* length of the complete translation table
                                in bytes */
   csttname cs_tname;        /* name of the translation table */
   ushort   cs_nref;         /* number of open file references */
   ushort   cs_pesc;         /* position of the escape prefix index */
   ushort   cs_nesc;         /* number of escape sequence prefixes */
   ushort   cs_pchset;       /* position of the character set index */
   ushort   cs_nchset;       /* number of character sets */
   ushort   cs_ptrchar;      /* position of the table of 16-bit
                                translated characters */
   ushort   cs_nextcs;       /* number of external character sets */
   ushort   cs_pextcs;       /* position of the table of escape
                                sequences used to select the external
                                character sets */
   ushort   cs_ttflag;       /* translation table flag bits */
};

typedef struct {
   char     dev[9];          /* terminal or printer device name */
   char     lang[7];         /* name of the language */
} csttname;

#if !KERNEL || defined_io

/*
 * Character set option flag bits for cs_ttflag
 */
#define CSEXTSO       1           /* if the external device codes use SO
                                     and SI to indicate that bit 7 is on
                                   */
#define CSINTERN      2           /* set for internal XSIS 058404 to
                                     XSIS 058404 translation tables */

#endif defined_io
```

Figure Cstermio-1. Csttbl Structure (Page 2 of 2)

Table Cstermio-1. Csttbl Structure Fields

Field	Description
cs_tnum	Number of this translation table. Identifies the translation table slot in kernel space for this translation table. The first slot is cs_tnum == 0. ● A CSGETTT with a cs_tnum of a translation table that currently occupies a slot returns a copy of the translation table from that slot. ● A CSGETTT with a cs_tnum of an empty slot sets errno == ENXIO. ● A CSSETTT with a cs_tnum of an occupied slot writes over the translation table in that slot unless the table is currently in use, in which case errno == EBUSY is set. ● A CSSETTT with a cs_tnum of an empty slot results in the translation table filling that slot. Unsuccessful ioctl CSGETTT and CSSETTT calls set the value of the pointer to the csttbl structure to -1, and errno is set appropriately.
cs_tlen	Size of the csttbl structure in bytes. See the CSMAXSIZ comment in Figure 4-2.
cs_tname	Name of the translation table (csttbl structure). The values for dev and lang in the structure type csttname are NULL-terminated strings representing TERM (terminal type) and LANG (user-language identifier), usually as contained in the execution environment.
cs_nref	Number of users of the translation table. Set by the terminal driver to indicate whether or not the translation table is in use.
cs_pesc	Offset of the first index structure of the csescix escape prefix in the csttbl.cs_tbl character array.
cs_nesc	Number of csescix escape prefix index structures in the csttbl.cs_tbl character array.

Table Cstermio-1. Csttbl Structure Fields (Continued)

Field	Description
cs_pchset	Offset of the first csescix character-set index structure in the csttbl.cs_tbl character array.
cs_nchset	Number of csescix character-set index structures in the csttbl.cs_tbl character array.
cs_ptrchar	Offset of the Char16Code translated characters table in the csttbl.cs_tbl character array.
cs_nextcs	Number of character sets declared as supported by the device to which this translation table applies. Equal to the number of entries in the cs_pextcs ushort array.
cs_pextcs	Offset of a ushort array of indexes to NULL-terminated strings that contain output sequences for device character-set selection.
cs_ttflag	cs_ttflag == CSEXTSO indicates that the device uses 7-bit codes with the ASCII SO (016) code to designate that the logical 8th bit is set on ensuing bytes until an ASCII SI (017) code is received. cs_ttflag == CSINTERN indicates that the translation operation is between internal character-set representations, not device-dependent code sequences.

Csescix Structure:

The csescix structure is used in mapping device-dependent inbound codes to Char16Code internal codes. This structure is shown in Figure Cstermio-2. Descriptions of the fields in the csescix structure are given in Table Cstermio-2.

```
/* description of an escape prefix index */
struct csescix {
    unsigned char   cs_esc[4];      /* escape sequence prefix */
    unsigned char   cs_esclo;       /* lowest valid character after
                                       prefix */
    unsigned char   cs_eschi;       /* highest valid character */
    ushort          cs_esctt;       /* position of the translation table
                                       table */
};
```

Figure Cstermio-2. Csescix Structure

Table Cstermio-2. Csescix Structure Fields

Field	Description
cs_esc[4]	NULL-terminated string of characters constituting a device-dependent escape sequence that precedes a character to be translated. An example of such a sequence is ESC N x, where ESC N is the escape sequence prefix, and x is a variable value.
cs_esclo	Lowest value of a range of characters subject to translation that follows the escape sequence prefix defined by cs_esc[4]. An example of this value is the lowest value for x in the sequence ESC N x.
cs_eschi	Highest value of a range of characters subject to translation that follows the escape sequence prefix defined by cs_esc[4]. An example of this value is the highest value for x in the sequence ESC N x.
cs_esctt	Offset of the csttent translation-table entry in the csttbl.cs_tbl character array associated with this escape sequence prefix.

Cscsix Structure:

The cscsix structure, shown in Figure Cstermio-3, is used in mapping Char16Code internal codes to device-dependent outbound codes. The fields in the cscsix structure are described in Table Cstermio-3.

```
/* description of a character set index */
struct cscsix {
        unsigned char   cs_csnum;       /* character set number */
        unsigned char   cs_cspfx[2];    /* possible prefix character(s)
                                           (character set 000) for
                                           accented characters and
                                           ligatures */
        unsigned char   cs_nlist;       /* number of list entries */
        ushort          cs_plist;       /* position of list entries */
        ushort          cs_cstt;        /* position of the translation
                                           table */
};
```

Figure Cstermio-3. Cscsix Stucture (Page 1 of 2)

```
#if !KERNEL || defined_io

/* used if cs_nlist == 0, so all values between cs_cslo and cs_cshi
   have translation tables.  Otherwise, cs_plist points to a list of
   the valid codes.
 */
#define cs_cslo(e)     (((e)->cs_plist)>>8)      /* lowest valid char. */
#define cs_cshi(e)     (((e)->cs_plist)&0xff)    /* highest valid char. */

#endif defined_io
```

Figure Cstermio-3. Cscsix Structure (Page 2 of 2)

Table Cstermio-3. Cscsix Structure Fields

Field	Description
cs_csnum	CharSet8 value designating the character set to which this cscsix structure applies. Legal values defined by the character-code standard are 000, 040 through 176, and 241 through 376.
cs_cspfx[2]	Contains (in cs_cspfx[0]) a character-set 000 accent character.
cs_nlist	Number of entries in the list of code sequences for outbound characters in the csttbl.cs_tbl character array. See the above definitions of cs_cslo and cs_cshi for the special meaning of cs_nlist == 0.
cs_plist	Offset of the list of valid accented letters and code sequences for outbound characters in the csttbl.cs_tbl character array associated with this character-set index when cs_nlist != 0.
cs_cstt	Offset of the csttent translation table entry in the csttbl.cs_tbl character array associated with this character-set index.

Csttent Structure:

The csttent structure and field descriptions are shown in Figure Cstermio-4.

```
/* description of translation table entry */
struct csttent {
        ushort        cs_tttyp:4;          /* entry type code */
        ushort        cs_ttval:12;         /* low bits of accent
                                              character */
};

#if !KERNEL || defined_io

/* the cs_tttyp values are: */
#define CS_NOCHG        0     /* value unchanged by translation */
/*                      1-7      number of 16-bit characters in entry */
#define CS-CSO          8     /* cs_ttchar in character set 000+cs_ttnib
                                */

#define CS_CS40         9     /* cs_ttchar in character set 040+cs_ttnib
                                */

#define CS_ACC          14    /* C0+cs_ttnib plus cs_ttchar in character
                                 set 000 */
#define CS_ERR          15    /* invalid character */

/* the cs_ttval field may be redefined as: */
#define cs_ttnib(e)           ((e)->cs_ttval >> 8)
#define cs_ttchar(e)          ((e)->cs_ttval & 0x0ff)

#endif defined_io

/* For cs_tttyp values of 1 through 7, the cs_ttval field contains a
   subscript into the array of ushort translated character
   sequences.  These 16-bit entries contain an 8-bit external
   character set number and an 8-bit character value (when cs_nextcs
   is greater than zero), an 8-bit character set number and an 8-bit
   character value (when CSINTERN is set in cs_ttflag), or an 8-bit
   escape sequence prefix number and an 8-bit char suffix.

   For cs_tttyp values of CS_CSO, CS_CS40 and CS_ACC, cs_ttchar
   contains a character value in the indicated character set.  For CS_
   ACC, cs_ttnib contains the low four bits of the accent character.
   For example, the csttent value 0xe841 represents a dieresis
   (0x00c8) followed by an uppercase A (0x0041).
 */
```

Figure Cstermio-4. Csttent Structure

A complete character-set translation table is shown in Figure Cstermio-5.

```
Header
   struct cstthdr

External character set index
   ushort []
                                           Strings

Escape prefix indexes
<sorted by prefix>
   struct csescix 0
          .
          .
          .
   struct csescix m-1

Escape sequence indexes
<sorted by character set, then accent>
   struct cscsix 0
          .
          .
          .
   struct cscsix n-1
                                           Strings

Translation table entries
<in each array from low to high character>
   struct csttent[] for csescix 0
          .
          .
          .
   struct csttent[] for csescix m-1
   struct csttent[] for cscsix 0
          .
          .
          .
   struct csttent[] for cscsix n-1

Translated character sequences
   ushort []

Strings
(used for character-set index lists and
external character-set escape sequences)
```

Figure Cstermio-5. Character-Set Translation Table Structure

Control of Terminal Character-Set Options:

There are two ways to control terminal I/O character-set translation options. You can use the cstty(1) command or the interface provided through the ioctl(2) system calls. Figure Cstermio-6 shows the structure, defined in <cstty.h>, that these calls use.

```
 /*
  * Character set option argument for CSGETO and CSSETO/OW/OF
  */
 struct      csopt {
             ushort     cs_options;     /* option bits */
             csttname   cs_name;        /* name of the character set
                                           translation table */
 };

 typedef struct {
       char             dev[9];    /* terminal or printer device name */
       char             lang[7];   /* name of the language */
 } csttname;
```

Figure Cstermio-6. Csopt Structure

The cs_options values have the following meanings, which are defined in <cstty.h>. See the description of cstty(1) options for more information on these options.

```
#define CSTRANS          0001       /* Select translation */
#define CS040            0002       /* Select character set 040 */
#define CSFMT7           0004       /* Select 7-bit SO/SI+SUB codes */
#define CST16            0010       /* Select 16-bit defined strings */
```

The values for dev and lang in the structure type csttname are NULL-terminated strings representing TERM (terminal type) and LANG (user language identifier), usually as contained in the execution environment.

Ioctl(2) calls to manage terminal I/O character-set translation tables have the form:

```
     ioctl (fildes, command, arg)
     cstbl *arg;
```

The commands using this form are:

CSGETTT   Get the character-set translation table parameters associated with the csttbl structure referenced by <arg>.

CSSETTT   Set the character-set translation table parameters associated with the structure referenced by <arg>.

Ioctl(2) calls to control terminal I/O character-set translation options for each active terminal have the form:

```
ioctl (fildes, command, arg)
IS_xctl *arg;
```

The commands using this form are:

CSGETC   Get the parameters associated with character-set translation for the terminal and store them in the IS_xctl structure referenced by <arg>.

CSSETC   Set the character-set translation parameters for the terminal from the structure referenced by <arg>. The change is immediate.

CSSETCW   Wait for the output to drain before setting the new parameters. Use this form when you change parameters that will affect output.

CSSETCF   Wait for the output to drain, then flush the input queue, and set the new parameters.

FILES

/dev/tty*	terminal or terminal-like device character special file
/etc/cs.term/*	terminal character-set translation source files installed at boot time
/usr/lib/cs.term/*	terminal character-set translation source files

SEE ALSO

stty(1), itt(1M), cstty(1), ioctl(2), termio(7)
Series 6000 International Support Package Reference Manual

NAME

fp – winchester, cartridge, and floppy disks

SYNOPSIS

```
/* MiniFrame only */
#include <sys/types.h>
#include <sys/gdisk.h>
#include <sys/gdioctl.h>
```

DESCRIPTION

The files **fp000** through **fp64F** and **rfp000** through **rfp64F** refer to slices on winchester, cartridge, and floppy disks. An **r** in the name indicates the character (raw) interface. On System 6600 the three hexadecimal digits are the file processor number, disk number, and slice number. The cartridge drive is disk 0 on file processor 0. On System 6300 the first digit is always 0, the second is the disk number (see below for standard disk number meanings) and the third is the slice number. At the lowest level, System 6600 handles disks quite differently from System 6300.

System 6600

System 6600 architecture greatly simplifies the operating system disk interface: RTOS manages disk initialization and low-level input/output; the operating system only accesses the disks to store and retrieve data. A disk special file is a reference to a RTOS disk file set aside specially for operating systems use. The RTOS file is called a *file system partition* and is created by *crup*(1M). The relationship between file system partitions and operating system special files is controlled by the RTOS file system configuration file, *[sys] <sys>configufs.sys.* It is described in the *System 6600 Administrator's Guide.*

System 6300

On System 6300, the operating system provides all disk facilities.

The disk numbers have the following standard meanings. Disk 0 is the winchester with the operating system. If there is a disk 1, it is either the second winchester or the removable hard disk cartridge drive (currently a System 6300 never has both). If there is a disk 2, it is the floppy drive.

It is worth discussing alien disks first. An alien disk is a disk not formatted by the System 6300; this means both disks formatted by other kinds of systems and disks never formatted at all. Slice 0 includes the entire disk, there is no bad block handling, and 512-byte sectors are assumed. Read and write the disk with the character special file only. To select a sector, seek to 1024 times the sector number. Use *ioctl* to describe the disk's format if it has one; see below. The *iv* utility uses this kind of interaction to format new disks.

System 6300 formats a disk with 512-byte physical sectors. Winchester and cartridge disks have 17 physical sectors per track. Floppies have 8 physical sectors on track 0, which is recorded in single density, and 16 physical sectors on all other tracks, which are recorded in double density.

Block input/output uses 1024-byte logical blocks. Winchester and cartridge disks have 8 logical blocks on each track, with the leftover physical block available as an alternate for a bad block. Floppies have 8 logical blocks on each track, with no bad block handling.

Logical block zero contains the *Volume Home Block*, which describes the disk. The following structure defines the volume home block.

```
struct vhbd {
        uint            magic;                  /* magic number */
        int             chksum;                 /* logical block checksum */
        struct gdswprt  dsk;                    /* description of this disk */
        struct partit   partab[MAXSLICE];       /* slice table */
        struct resdes {  /* reserved area special files */
                daddr_t blkstart;/* start logical blocks */
                ushort  nblocks;/* length in logical blocks */
        }               resmap[8];              /* reserved area files */
        char            fpulled;                /* pulled flag */
        long            time                    /* time last came on line */
        short           rwcwpccyl               /* reduced write current/
                                                 /*write precompensation value */
        char            minires[74];            /* reserved */
        char            sysres[292];
        struct mntnam mntname[MAXSLICE]; /* reserved */
        char            userres[256];           /* reserved */
};

struct gdswprt {
        char    name[6]; /* disk name */
        ushort cyls;     /* cylinders */
        ushort heads;    /* heads */
        ushort psectrk;  /* physical sectors/track */
        ushort pseccyl;  /* physical sectors/cylinder */
        char    flags;   /* disk type flags */
        char    step;    /* stepper motor rate */
        ushort sectorsz; /* physical sector size/bytes */
};

struct partit {                         /*partition table*/
        union {
                uint strk;              /* start track number (new style) */
                struct {
                        ushort strk   /* start track # */
                        ushort nsecs;/* # logical blks available to user */
                } old;
        } sz;
};

struct mntnam {
        char name[MNAMSIZ];
}
```

If a volume home block is valid, *magic* is equal to VHBMAGIC and the 32-bit sum of the volume home block's bytes is 0xFFFFFFFF (-1); *chksum* is the adjustment that makes the sum come out right.

Dsk describes the peculiarities of the disk, including deliberate deviations from the System 6300 standard. *Dsk.flags* the bitwise or of zero or more of the following constants:

FPDENSITY	If on the disk is double density; if off the disk is single density.
FPMIXDENS	If off, FPDENSITY specifies the density of the first track; if on, the first track is single density regardless of **FPDENSITY**.
HITECH	If on, head select bit 3 is valid; if off, reduced write current is valid.
NEWPARTTAB	If off, the old stype slice (partition) table is in use; if on, the new style slice table is in use.
RWCPWC	If on, set reduced write current/write precompensation. **HITECH** selects write precompensation.
EXCHANGEABLE	If on, the disk is a floppy or removable hard disk cartridge. If off, the disk is a winchester.

Dsk.step specifies a stepper motor rate; currently this field must be 0.

Partab divides the disk into slices (partitions).

Fpulled indicates whether an exchangeable disk was properly removed from the drive. The system sets this field to 1 when the disk is inserted in the drive. To clear *fpulled*, run *dismount*(1M); see that entry.

Mntname, *minires*, and *userres* are reserved for future use.

Resmap describes the files that share Slice 0 with the Volume Home Block. Provision is made for eight such files, but only five have been assigned slots in *resmap*. Each *resmap* entry gives the starting location (logical block number) and length (logical blocks). A length of zero indicates that the file is not provided. The first five entries in *resmap* describe:

1. The loader. When System 6300 is reset or turned on, the boot prom loads the loader into address 0x70000 and jumps execution to it. The function of the loader is to search for and load a program that will boot the system. The loader searches the floppy, the cartridge disk, and the winchester, in that order. On each disk, the loader first checks for a standalone program. If the disk lacks a standalone program, the loader checks for an operating system kernel, which must be an operating system executable object file called /unix in the file system in slice 1. When the loader locates an appropriate program, it preserves the crash dump table (0x70000 – 128), loads the program it found at the address it was linked at (0x0 if unknown) and executes it. If no disk contains an appropriate file, the loader continues searching until an appropriate disk is inserted.

2. The bad block table, which always begins at logical block 1 of the disk. Each logical block in the bad block table consists of a four-byte checksum followed by 127 bad block cells. The checksum is a value that makes the 32-bit sum of the logical block be 0xFFFFFFFF (–1). A bad block cell is defined by the following structure.

   ```
       struct bbcell {
               ushort cyl;
               ushort badblk;
               ushort altblk;
               ushort nextind;
       }
   ```

   A single sequence of numbers, starting from zero, identifies the checksums and cells. In each cell in use, *cyl* identifies a cylinder that contains the bad block; *badblk* physical block offset within the cylinder of the bad block; *altblk* identifies the cylinder that contains the alternate block; *nextind* identifies the next cell for a bad block on the same cylinder or is zero if this is the last one.

3. The dump area. After Reset or Suicide, the Boot prom dumps processor registers, the memory map, a crash dump block, and the contents of physical memory, until it runs out of room in the dump area.

4. The down load image area. The down load images are described by a table at the beginning of the area. The area is described by the following array.

   ```
   struct dldent {
           short d_strt;
           short d_sz;
   } dldent[DLDSZ];
   ```

   The image number is the index for *dldent*. *D_strt* is the offset in bytes of the image from the beginning of the down load image area; *d_sz* is the size in bytes of the image. Image number *xxx* is the same as the System 6600 RTOS file

[sys]<**sys**>**ws***xxx*>**sysimage.sys**, where *xxx* must be expressed in three digits.

5. A bootable program, usually a diagnostic. This is the program the loader considers a substitute for the **/unix** file. The program must be in *a.out*(4) format with magic number 407 or be a simple memory image.

   If the fifth entry in *resmap* has a zero address but a nonzero length, the loader looks at the beginning of slice 1 for the program.

Slice 0 is called the Reserved Area. Only the volume home block and the files described by *resmap* can be in the Reserved Area. A System 6300-formatted disk used by a working system certainly has at least one more slice.

System 6300 *ioctl* system calls use the following structure.

```
struct gdctl {
        unsigned short status;
        struct gdswprt params;
        short           dsktype;
}
```

Status is the bitwise or of the following constants.

VALID_VHB	A valid Volume Header Block has been read.
DRV_READY	The disk is on line.
PULLED	Last removal of disk from drive was not preceded by proper dismount.

Params is a *gdswprt* structure, the same type used in the volume header block.

Dsktype is equal to **GD_WIN** (winchester), **GD_SYQ** (cartridge), or **GD_FLP** (floppy).

System 6300 operating system understands the following disk *ioctl* calls.

ioctl(fd, GDIOCTYPE, 0)
: Returns **GDIOC** if *fd* is a file descriptor for a disk special file.

ioctl(fd, GDGETA, gdctl_ptr)
: *Gdctl_ptr* is a pointer to a *gdctl* structure. *Ioctl* fills the structure with information about the disk.

ioctl(fd, GDSETA, gdctl_ptr)
: *Gdctl_ptr* is a pointer to a *gdctl* structure. *Ioctl* passes the description of the disk to the disk driver. This is primarily meant for reading disks created by other kinds of computers.

ioctl(fd, GDFORMAT, ptr)
: *Ptr* points to formating information. The disk driver formats a track.

ioctl(fd, GDDISMNT)
: *Ioctl* informs the driver that the user intends to remove the disk from the drive. When this system call successfully returns, the driver has flushed all data in the buffer cache and waited for all queued transfers to complete. The last transfer is to write out the Volume Home Block with the *fpulled* flag cleared. Once this call returns the drive is inaccessible until a new disk is inserted.

SEE ALSO

crup(1M), mknod(1M), ofcopy(1M), ioctl(2).

NAME

lp – parallel printer interface

DESCRIPTION

Lp is an interface to the parallel printer channel. Bytes written are sent to the printer. Opening and closing produce page ejects. Unlike the serial interfaces (*termio*(7)), the *lp* driver never prepends a carriage return to a new line (line feed). The *lp* driver does have options to filter output, for the benefit of printers with special requirement. The driver also controls page format. Page format and filter options are controlled with *ioctl*(2):

```
#include <sys/lprio.h>
ioctl(fildes, command, arg)
```

where *command* is one of the following constants:

LPRSET — Set the current page format from the location pointed to by *arg*; this location is a structure of type **lprio**, declared in the header file:

```
struct lprio {
        short ind;
        short col;
        short line;
}
```

Arg should be declared as follows:

```
struct lprio *arg;
```

Ind is the page indent in columns, initially 4. *Col* is the number of columns in a line, initially 132, *Line* is the number lines on a page, initially 66. A newline that extends over the end of a page is output as a formfeed. Lines longer than the line length minus the indent are truncated.

LPRGET — Get the current page format and put it in the **lprio** structure pointed to by *arg*.

LPRSOPTS — Set the filter options from *arg*, which must be of type **int**. *Arg* should be the logical or of one or more of the following constants, defined in the header file:

Constant	*Value*	*Meaning*
LPNOBS	4	No back space. Set this bit if the printer cannot properly interpret backspace characters. The driver uses carriage return to produce equivalent overstriking.
LPRAW	8	Raw output. Set this bit if the driver must not edit output in any way. The driver ignores all other option bits in the minor device number.
LPCAP	16	Capitals. This option supports printers with a "half-ASCII" character set. Lowercase is translated to uppercase. The following special characters are translated: { to (-, } to)-; ` to '-; | to !-; ~ to ^-.

LPNOCR	32	No Carriage Return. This option supports printers that do not respond to a carriage return (character 0D hexadecimal). Carriage returns are changed to newlines. If No Newline is also set, carriage returns are changed to form feeds.
LPNOFF	64	No Form Feed. This option supports printers that do not respond to a form feed (character 0C hexadecimal). Form Feeds are changed to newlines. If No Newline is also set, form feeds are changed to carriage returns.
LPNONL	128	No Newline. This option supports printers that do not respond to a newline (character 0A hexadecimal). Newlines are changed to carriage returns. If No Carriage Return is also set, newlines are changed to form feeds.

Setting all three of No Carriage Return, No New Line, and No Form Feed has the same effect as setting none of them.

LPRGOPTS Get the current state of the filter options and put them in *arg*, which must be an **int**.

On System 6300, the device's minor device number can also be used to set and get filter options. The values are the same as for the LPRSOPTS and LPRGOPTs calls. This convention does not work on System 6600 because that system can have more than one parallel printer.

FILES

/dev/lp

SEE ALSO

lpr(1), lpset(1).

NAME

mem, kmem – core memory

DESCRIPTION

Mem is a special file that is an image of the core memory of the operating system-based processor board. It may be used, for example, to examine, and even to patch the system.

Byte addresses in *mem* are interpreted as memory addresses. References to non-existent locations cause errors to be returned.

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present.

The file *kmem* is the same as *mem* except that kernel virtual memory rather than physical memory is accessed.

FILES

On System 6300, /dev/mem, /dev/kmem.

On System 6600 /dev/mem*xx*, /dev/kmem*xx*, where *xx* is the two-digit processor number.

BUGS

*Mem***xx** is not currently implemented on System 6600.

NAME

mt – interface for magnetic tape

DESCRIPTION

This interface provides access to all magnetic tape drives. The naming conventions on System 6600 and System 6300 differ in the following ways:

- On System 6600, **mt*x*** is the block device with rewind on close for drive *x*. To get the no-rewind device, prepend **n**; to get the raw (character) device, prepend **r**; and to get the no-rewind on close, raw device, prepend **nr**.

  There can be up to four drives, any of which can be built-in QIC (quarter-inch cartridge) drives or external drives controlled by a Storage Processor. The connection between drives and drive numbers is in the file system configuration file, under RTOS: see the *System 6600 UNIX-Derived Operating System Administrator's Guide.*
- A System 6300 system has at most a single QIC drive. There are currently no block devices. To get the raw, rewind on close device, use **rmt0**. To get the raw, no-rewind on close device, use **rmt4**.

Tape files are separated by tape marks, also known as EOFs. Closing a file open for writing writes one tape mark on a QIC drive and two tape marks on other drives; if the device was no-rewind, the tape is left positioned just after the single QIC tape mark or between the two marks. If the file was a no-rewind file, reopening the drive for writing overwrites the second mark, if there is one, and creates another tape file. Thus on a QIC drive, a single tape mark separates the tape files and ends the tape; on other drives, a single tape mark separates the tape files and a double mark ends the tape.

Here are summaries of block and character device features:

- The block devices read and write only 1024-byte physical blocks; *reads* and *writes* of other sizes are resolved into 1K physical I/O. *Seeks* are ignored on QIC drives. On other drives *seeks* are allowed, but once the file is opened, reading is restricted to between the last write and the next tape mark. Reading the tape mark produces a zero-length *read* and leaves the tape positioned after the tape mark; if the file is a no-rewind file, the program can access the next tape file by closing the device and then reopening or opening another device for the same drive.
- On the raw devices, each *read* or *write* reads or writes the next physical block. A *read* must match the size of a normal tape block. The size of a *write* determines the size of the next block; *Write* sizes must be a multiple of 512 on QIC drives, a multiple of 2 on other drives. *Read/write* buffers must begin on an even address; this is the same alignment as **short**. Seeks are ignored. Reading a tape mark produces a zero-length *read* and leaves the tape positioned after the mark; the program can, without closing the device, read the next tape file.

FILES

/dev/mt?
/dev/nmt?
/dev/rmt?
/dev/nrmt?

WARNING

A nondata error cannot be recovered from except by closing the device.

A QIC tape has no special mark for end of tape, as opposed to end of file.

NAME

null – the null file

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

NAME

termio – general terminal interface

DESCRIPTION

Operating systems use a single interface convention for all RS-232 and cluster (RS-422) terminals, although cluster terminals do not use all the features of the convention. The convention is almost completely taken from the UNIX System V interface for asynchronous terminals.

Three kinds of terminals use this convention:

- RS-232 terminals connected to channels on the System 6600 or System 6300 itself.
- Cluster terminals. Generally a cluster channel supports more than one terminal and some terminals are indirectly connected through other terminals. Cluster terminals use the same interface as directly connected RS-232 terminals, except that hardware control operations are meaningless on cluster terminals.
- Local RS-232 terminals. These are connected to RS-232 channels on cluster terminals. They actually use the cluster terminal's RS-422 channel to communicate with the host computer system, but work like regular RS-232 terminals. Currently, they only work on System 6300.

A single naming convention applies to regular RS-232 and cluster terminals; a second, related, convention applies to local RS-232 terminals. A direct RS-232 or cluster terminal has a name of the form **tty***xxx*, where *xxx* is the terminal's number expressed in three digits. A local RS-232 terminal has a name of the form **tp***cxxx* where *c* is the RS-232 channel number (**1** or **2**), and *xxx* is the accomodating cluster terminal's terminal number expressed in three digits.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by *getty* and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *control terminal* for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork*(2). A process can break this association by changing its process group using *setpgrp*(2).

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 only lost when the system's character input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. When the input limit is reached, all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a newline (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, the character generated by a TM6000 Terminal BACK SPACE key (ASCII BS, Control-H on most terminals) erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the character @ kills (deletes) the entire input line, and optionally outputs a newline character. Both these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). In this case the escape character is not read. The erase and kill characters may be changed.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (Rubout or ASCII DEL; generated by a TM6000 Terminal DELETE key) generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally, 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).

QUIT (Control- or ASCII FS; generated by a TM6000 Terminal CODE-CANCEL key) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.

ERASE (Control-h or ASCII BS; generated by a TM6000 Terminal BACKSPACE key) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.

KILL (@) deletes the entire line, as delimited by a NL, EOF, or EOL character.

EOF (Control-d or ASCII EOP; generated by a TM6000 Terminal CTRL-EXIT key) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a newline, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It can not be changed or escaped.

EOL (ASCII NUL) is an additional line delimiter, like NL. It is not normally used.

STOP (Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (Control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL may be changed to suit individual tastes. The ERASE, KILL, and EOF characters may be escaped by a preceding \ character, in which case no special function is done.

When the carrier signal from the data-set drops, a *hangup* signal is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any subsequent read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. 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 has drained down to some threshold, the program is resumed.

Several *ioctl*(2) system calls apply to terminal files. The primary calls use the following structure, defined in **<termio.h>**:

```
#define  NCC       8
struct   termio {
         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 chars */
};
```

The special control characters are defined by the array *c_cc*. The relative positions for each function are as follows:

0	INTR
1	QUIT
2	ERASE
3	KILL
4	EOF
5	EOL
6	reserved
7	reserved

The *c_iflag* field describes the basic terminal input control:

IGNBRK	0000001	Ignore break condition.
BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
PARMRK	0000010	Mark parity errors.
INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.
INLCR	0000100	Map NL to CR on input.
IGNCR	0000200	Ignore CR.
ICRNL	0000400	Map CR to NL on input.
IUCLC	0001000	Map upper-case to lower-case on input.
IXON	0002000	Enable start/stop output control.
IXANY	0004000	Enable any character to restart output.
IXOFF	0010000	Enable start/stop input control.

If IGNBRK is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if BRKINT is set, the break condition will generate an interrupt signal and flush both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

If PARMRK is set, a character with a framing or parity error which is not ignored is read as the three character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error which is not ignored is read as the character NUL (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper-case alphabetic character is translated into the corresponding lower-case character.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read. If IXANY is set, any input character, will restart output which has been suspended.

If IXOFF is set, the system will transmit START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all bits clear.

The *c_oflag* field specifies the system treatment of output:

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lower case to upper on output.
ONLCR	0000004	Map NL to CR-NL on output.
OCRNL	0000010	Map CR to NL on output.
ONOCR	0000020	No CR output at column 0.
ONLRET	0000040	NL performs CR function.
OFILL	0000100	Use fill characters for delay.
OFDEL	0000200	Fill is DEL, else NUL.
NLDLY	0000400	Select new-line delays:
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays:
TAB0	0	
TAB1	0004000	
TAB2	0010000	
TAB3	0014000	Expand tabs to spaces.
BSDLY	0020000	Select backspace delays:
BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	
FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

The delay bits specify how long 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. If OFILL is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL, otherwise NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

new-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the new-line delays. If OFILL is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.18 seconds. If OFILL is set, delay type 1 transmits one or two fill characters, and type 2 and 3 two fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.04 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, delay type 1 transmits zero or two fill characteres and delay type 2 transmits 1 fill character.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

The *c_cflag* field describes the hardware control of the terminal (not used on cluster terminals):

CBAUD	0000017	Baud rate:
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B1200	0000011	1200 baud
B1800	0000012	1800 baud
B2400	0000013	2400 baud
B4800	0000014	4800 baud
B9600	0000015	9600 baud
EXTA	0000016	19200 baud
EXTB	0000017	External clock.
CSIZE	0000060	Character size:
CS5	0	5 bits
CS6	0000020	6 bits
CS7	0000040	7 bits
CS8	0000060	8 bits
CSTOPB	0000100	Send two stop bits, else one.
CREAD	0000200	Enable receiver.
PARENB	0000400	Parity enable.
PARODD	0001000	Odd parity, else even.
HUPCL	0002000	Hang up on last close.
CLOCAL	0004000	Local line, else dial-up.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal will not be asserted. Normally, this will disconnect the line. For any particular hardware, impossible speed changes are ignored. EXTB specifies external clocking.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stops bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used.

If CREAD is set, the receiver is enabled. Otherwise no characters will be received.

If HUPCL is set, the line will be disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. Otherwise modem control is assumed.

The initial hardware control value after open is B9600, CS8, CREAD, HUPCL.

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	0000001	Enable signals.
ICANON	0000002	Canonical input (erase and kill processing).
XCASE	0000004	Canonical upper/lower presentation.
ECHO	0000010	Enable echo.
ECHOE	0000020	Echo erase character as BS-SP-BS.
ECHOK	0000040	Echo NL after kill character.
ECHONL	0000100	Echo NL.
NOFLSH	0000200	Disable flush after interrupt or quit.

If ISIG is set, each input character is checked against the special control characters INTR and QUIT. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g. 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read will not be satisfied until at least MIN characters have been received or the timeout value TIME has expired. This allows fast bursts of input to be read efficiently while still allowing single character input. The VMIN and VTIME values are stored in the position for the VEOF and VEOL characters respectively. The time value represents tenths of seconds.

If XCASE is set, and if ICANON is set, an upper-case letter is accepted on input by preceding it with a \ character, and is output preceded by a \ character. 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**.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which will clear the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character will be echoed after the kill character to emphasize that the line will be deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character will be echoed even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit and interrupt characters will not be done.

The initial line-discipline control value is all bits clear.

The primary *ioctl*(2) system calls have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETA	Get the parameters associated with the terminal and store in the *termio* structure referenced by **arg**.
TCSETA	Set the parameters associated with the terminal from the structure referenced by **arg**. The change is immediate.
TCSETAW	Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.
TCSETAF	Wait for the output to drain, then flush the input queue and set the new parameters.

Additional *ioctl*(2) calls have the form:

```
ioctl (fildes, command, arg)
int arg;
```

The commands using this form are:

TCSBRK	Wait for the output to drain. If *arg* is 0, then send a break (zero bits for 0.25 seconds).
TCXONC	Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output; if 2, transmit XOFF; if 3, transmit XON.
TCFLSH	If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

FILES

/dev/tty??? /dev/tp????

SEE ALSO

stty(1), ioctl(2), tp(7), tty(7).

WARNING

The default value for ERASE is backspace rather than the historical #.

BUGS

Local RS-232 terminals do not currently provide hangup (B0), draining, flushing, or delay.

NAME

tp – controlling terminal's local RS-232 channels

DESCRIPTION

The **tp** devices accesses the RS-232 channels on the controlling terminal. The terminal must be a cluster terminal configured to permit use of the local RS-232 channels (see *termio*(7). Just as **/dev/tty** permits a process to conveniently access its process group's controlling terminal (see *tty*(7)), **/dev/tp1** and **/dev/tp2** access the controlling terminal's RS-232 channels without reference to the terminal number. This is convenient for accessing the user's local hardware, such as a telephone with an RS-232 interface.

SEE ALSO

tty(7).

NAME

tty – controlling terminal interface

DESCRIPTION

The file **/dev/tty** is, in each process, a synonym for the control terminal associated with the process group of that process, if any. It is useful for programs or shell sequences that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

If the terminal is under window management, a process group is controlled by a specific window and I/O on **/dev/tty** is directed to that window. A terminal can control one process group in each window. See *window*(7).

FILES

/dev/tty

SEE ALSO

tp(7), window(7).

NAME

window – window management primitives

SYNOPSIS

#include <sys/window.h>

DESCRIPTION

Window managment (*wm*(1)) provides a superset of windowless terminal features. This entry describes terminal file features special to window management. Window management features are designed not to interfere with programs that do not know about window management. Such design includes simple extensions to the UNIX System's standard concepts of file descriptor and control terminal.

- Each terminal file descriptor has an associated window number, a small positive integer that identifies a window. A window number is the most primitive way to refer to a window, and should not be confused with the window ID used by window management subroutines. A new window gets the smalled window number not already in use. Closing a window frees its number for possible assignment to a later window. Output and control calls on the file descriptor apply only to the descriptor's window; input calls succeed only when the window is active.

  A file descriptor created by a *dup*(2) or inherited across a *fork*(2) inherits the original descriptor's window number. All the file descriptors in such a chain of inheritance, provided they belong to processes in the same process group, are affected when *ioctl* changes the window number of any of them.

- When a process group's control terminal is under window managment, the process group is actually controlled by a particular window. Such can have more than one process group, each controlled by a different window. Keyboard-generated signals (*interrupt* and *quit*) go to the process group controlled by the active window.

When the user creates a new window by using the RESET key, the window manager forks a process for that window. The new process inherits file descriptors for standard input (0), standard output (1), and standard error (2) that are associated with the new window. The new process is leader of a process group controlled by the new window.

Programs that create and use windows use window management *ioctl*(2) calls. Such calls take the form

```
ioctl (fildes, command, arg)
struct wioctl *arg;
```

Fildes is a file descriptor for terminal and window affected, *command* is a window management command (see below) *arg* is a pointer to the following structure, declared in **<sys/window.h>**:

```
#define NWCC 2

struct wioctl {
        wndw_t              wi_dfltwndw;
        wndw_t              wi_wndw;
        slot_t wi_mycpuslot;
        slot_t wi_destcpuslot;
        port_t wi_bport;
        char   wi_dummy;
        unsigned char wi_cc[NWCC];
};
```

Window management *ioctl* calls get (WIOCGET) and set (WIOCSET and WIOCSETP) terminal attributes described in the *wioctl* structure:

wi_dfltwndw	The window number for the process's default window. If the process does an *open* on **/dev/tty**, the new file descriptor is associated with the default window.
wi_wndw	The window number for the window that *fildes* (*ioctl's* first parameter) is associated with.
wi_mycpuslot	The slot number of the process's host processor. (Not settable. Not meaningful on System 6300.)
wi_destcpuslot	The slot number of the processor that drives the terminal. (Not settable. Not meaningful on System 6300.)
wi_bport	The terminal's Cluster Processor or Terminal Processor channel number (Not settable. Not meaningful on System 6300.)
wi_cc	Not used by the operating system kernel. A value supplied by a WIOCSET or WIOCSETP is stored in a place associated with window *wp_wndw*. A subsequent WIOCGET on the same window retrieves the information.

Here are the window management *ioctl* commands:

WIOCGET — Get information on calling process and file descriptor *fildes*. Fill in *arg*.

WIOCSET — Set values for calling process and file descriptor *fildes* from information in *arg*. Has no effect on process group-control terminal relationship.

WIOCSETP — Set values for calling process and file descriptor *fildes* from information in *arg*. The window specified in *arg->wi_wndw* becomes the process's group's controlling terminal provided the following:

- The calling process is the process group leader.
- The process group is not currently controlled by another window on this or any other terminal.
- The specified window is not already a control window.

WIOCLRP — Only valid executed by process group leader. The process group ceases to have a control terminal or window and the control terminal/window ceases to control any process group. The process group is free to find another control terminal/window, and the old control terminal/window is free to become the control terminal/window for another process group.

WIOCCLUSTER
Ioctl returns 1 if and only if the terminal is a cluster terminal.

WIOCDIRECT (System 6300 only.) Enable direct sending of terminal IPC requests.

WIOCUNDIRECT
(System 6300 only.) Disable direct sending of terminal IPC requests.

An *open* on a terminal special file other than **/dev/tty** (for example, **/dev/tty000**) produces a file descriptor for the lowest-numbered open window. *Ioctl* can move this file descriptor to any window.

An *open* can also obtain a controlling terminal/window. The requirements are the same as for WIOCSETP.

FILES

/dev/tty – control terminal
/dev/tty??? – terminals

SEE ALSO

stty(1), wm(1), dup(2), fork(2), ioctl(2), open(2), wmgetid(3X), wmlayout(3X), wmop(3X), wmsetid(3X), termio(7), tty(7).

WARNINGS

WIOCDIRECT and WIOCUNDIRECT are required by the operating system. Their use by user programs is inadvisable.

Use these features in as standard and conservative a way as possible. The best way to enforce standards is to use window management through the library calls described in Section 3.

USER'S COMMENTS

Series 6000 Operating System Reference Manual, Volume II
Stock Number: 87601845C 44209-02

HELP!

Help us help you! Please take the time to complete this form and send it to us. If you do, you may see some of your own contributions in the next manual you obtain from us.

- Does this manual provide the information you need? ☐ Yes ☐ No
  - What is missing?

- Is the manual accurate? ☐ Yes ☐ No
  - What is incorrect? (Be specific.)

- Is the manual written clearly? ☐ Yes ☐ No
  - What is unclear?

- What other comments can you make about this manual?

- What do you like about this manual?

- On a scale of 1 to 10, how do you rate this manual? Low 1 2 3 4 5 6 7 8 9 10 High

- Was this manual difficult to obtain? ☐ Yes ☐ No

Please include your name and address if you would like a reply.

Name ______________________________
Company ____________________________
Address _____________________________
____

No postage required if mailed within the USA.

- What is your occupation?

☐ Programmer	☐ Operator	☐ Manager
☐ Systems Analyst	☐ Instructor	☐ Customer Engineer
☐ Engineer	☐ Student	☐ Other ________

- How do you use this manual?

☐ Reference Manual	☐ Introduction to the Subject
☐ In a Class	☐ Introduction to the System
☐ Self Study	☐ Other ________

fold **fold**

FIRST CLASS
Permit No. 194
Cupertino,
California

BUSINESS REPLY MAIL
No Postage Necessary if Mailed in the United States

Postage will be Paid by . . .

MOTOROLA INC.
10700 N. De Anza Blvd.
Cupertino, CA 95014

Attention: Technical Services, MS 32-2G2

Cut Along Here

fold **fold**

Staple Here

B1847A