NAME

intro – introduction to special files

DESCRIPTION

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

BUGS

While the names of the entries *generally* refer to DEC hardware names, in certain cases these names are seemingly arbitrary for various historical reasons.

NAME

dh, dz — asynchronous multiplexers

DESCRIPTION

Each line attached to a DH-11 or DZ-11 communications multiplexer behaves as described in *tty*(4). Input and output for each line may independently be set to run at any of 16 speeds; see *ioctl* (2) for the encoding. (For DZ-11 lines, output speed is always the same as input speed. The *200* speed and the two externally clocked speeds *(exta, extb)* are missing on the DZ-11.)

FILES

/dev/ln*

SEE ALSO

ioctl(2), tty(4)

NAME

dn − DN-11 ACU interface

DESCRIPTION

The **dn??** files are write-only. The permissible codes are:

0−9	dial 0-9
*****	dial *
#	dial #
−	4 second delay for secondary dial tone (for use with older ACU's)
e	end-of-number
w	forces ACU to wait for secondary dial tone (for use with newer ACU's)
f	flashes switchhook if a flasher card is installed (HOBIS application)
d	drop call
r	return from write leaving call request on

The entire telephone number should be presented in a single *write* system call unless a flasher card is used in special applications.

A write call returns the number of digits sent when the answer tone is detected and control is given to the data set. If the digit string is terminated with an 'e', the write call returns the number of digits sent without waiting for answer tone. An error condition causes a **−1** to be returned.

The *dn* interface is divided into two mutually exclusive functions. The first function provides an ACU interface when the ACU is associated with only one data set. The second function provides an ACU interface when the ACU is associated with multiple data sets (maximum of 12). Minor device numbers in the range of 0 − 15 inclusive are interpreted by the *dn* to be of the single ACU/single data set arrangement. Minor device numbers greater than 15 are interpreted by the *dn* to be a single ACU sharing up to 12 data sets. *Dn*'s with a minor device number greater than 15 generate a data set select code to the ACU interface prior to returning from an open. The following algorithm must be used in assigning minor device numbers to insure that proper data set select codes are generated by the ACU.

(minor device number modulo 16) + 1 = data set select number

NOTE: Data set select numbers 7,8,15, and 16 are illegal and therefore so are the minor device numbers that generate them.

Also, it should be noted that minor device numbers in the range of 16−31 inclusive will actually perform auto calling functions using the first physical *dn* in the system, those in the range of 32−47 inclusive will perform auto calling using the second physical *dn* in the system, and so on. This implies that either minor device 0 is implemented for a single ACU/single data set arrangement or minor devices 16−31 are implemented to provide data set selection for up to 12 data sets associated with the *dn* that normally would be accessed via minor device 0.

FILES

/dev/dn?

BUGS

The reception of a signal which is being ignored during an open or write of the *dn* may cause an incorrect phone number to be dialed without returning an error indication.

SEE ALSO

dh(4)

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; the record is truncated if the read request is for less than the record's length.

FILES

/dev/error

SEE ALSO

errdemon(1)

NAME

hp − RP04/RP05/RP06 moving-head disk

DESCRIPTION

The files **hp0**, ..., **hp31** refer to sections of the RP04/RP05/RP06 disk drive 0. The files **hp32**, ..., **hp63** refer to drive 1, etc. This slicing allows the pack to be broken up into more manageable pieces.

A sample of the origin and size of the sections on each drive are as follows:

NCYL = 418 (22 * 19)

blocks	offset	section	
120*NCYL	0	hp0	overlaps 8,9,10,11
120*NCYL	120	hp1	overlaps 12,13
120*NCYL	240	hp2	
120*NCYL	360	hp3	
120*NCYL	480	hp4	
120*NCYL	600	hp5	
95*NCYL	720	hp6	
	(end of RP06)		
51*NCYL	360	hp7	
	(end of RP04/RP05)		
12*NCYL	0	hp8	util
11*NCYL	12	hp9	
97*NCYL	23	hp10	
0	0	hp11	spare
65*NCYL	120	hp12	source
55*NCYL	185	hp13	rootdev
-	-	-	
-	-	-	
	etc.		

Blocks are the number of cylinders assigned to a section of the disk times 418 blocks per cylinder. Offset is in multiple of cylinders and indicates where each section of the disk begins. Section refers to the *hp* number found in **/dev/hp***. It should be noted that **/dev/hp8** and **dev/util** are linked together and therefore, describe the same section of the disk. Also and **/dev/rootdev** are linked together and both refer to the root file system. This layout should be made with discretion to allow for convenient backups (overlays) and future expansion.

The *hp* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw *HP* files begin with **rhp** and end with a number which selects the same disk section as the corresponding *hp* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *lseek* calls should specify a multiple of 512 bytes.

By convention, programs never access the physical names **dev/hp*** or **/dev/rhp***, but access the logical names such as **/dev/musr** or **/dev/rmusr** instead. These logical names are linked by the system administrator to the physical device names.

FILES

/dev/hp*
/dev/rhp*

NAME

kl — KL-11 or DL-11 asynchronous interface

DESCRIPTION

The discussion of typewriter I/O given in *tty(4)* applies to these devices.

Since they run at a constant speed, attempts to change the speed via *ioctl (2)* are ignored.

The on-line console typewriter is normally interfaced using a KL-11 or DL-11.

The system console can be accessed via three different methods. They are as follows:

Ln 00 is an entry in the lines file to allow a *who (1)* command to identify the line. This entry is here for historical reasons only.

Systty is the real system console that is physically attached to the KL-11 or DL-11. This is where all the system error messages are printed.

Syscon is the virtual console that can be linked to another line for remote reboot. *Init (1M)* and *reboot (1M)* talk to this device.

Normally, *systty*, *syscon* and *ln 00* are all linked together. *Telinit (1M)* will change this arrangement if changing run level to level 7 causes the current terminal to become *syscon*.

FILES

/dev/ln00
/dev/systty
/dev/syscon

SEE ALSO

tty(4), who(1), telinit(1), reboot(1), init(1M)

BUGS

Modem control for the DL-11E is not implemented.

NAME

kmc − KMC11/DMC11 microprocessor

DESCRIPTION

The files **kmc?** are used to manipulate the KMC11 or DMC11 microprocessors. The only KMC11 currently supported is the KMC11-B. The device handler provides the basic mechanism needed to load, run, and debug programs on the microprocessor.

The open is exclusive; at most one open at a time is allowed. The first open can determine whether the microprocessor is a KMC11 or DMC11 by testing for bit 8 of the microprocessor memory address register; however, this test is disabled in the current implementation, and a KMC11-B is assumed.

Addresses 0−8191 (2047 for the DMC) reference the 4096 (1024 for DMC) words of instructions in the control memory of the microprocessor. For the KMC11, they may be read or written; for the DMC11, they are read-only. This portion is word oriented, that is, the address and byte count must be even.

Addresses 8192-12287 (2048-2303 for DMC) reference the 4096 (256 for DMC) bytes of data in the data memory of the KMC11. The data portion may be read or written with no restrictions on addressing.

The ioctl function is used to provide access to the basic microprocessor capabilities.

```
ioctl(kmcfd, KIOCSETD, argkmcbuf)
struct  {
        int     code;
        int     *csr;
        int     value;
} *kmcbuf;
```

The pointer *csr* contains the address of a 4 word buffer for the UNIBUS Control and Status Registers associated with the microprocessor. The value of *code* determines the function:

1 single step and return CSRs in *csr*.
2 maintenance step: execute *value* and then return CSRs.
3 return CSRs.
4 stop: clear the run bit.
5 reset: set then clear the master clear bit.
6 run: set the run bit and set the software state to *value* and running.
7 line unit maintenance: set the line unit bits from *value*.

FILES

/dev/kmc?

SEE ALSO

kasb(1), kunb(1).

NAME

lp − line printer

DESCRIPTION

Lp provides the interface to any of the standard Digital Equipment Corporation line printers. When it is opened or closed, a suitable number of page ejects is generated. Bytes written are printed.

An internal parameter within the driver determines whether or not the device is treated as having a 96- or 64-character set. In half-ASCII mode, lower case letters are turned into upper case and certain characters are escaped according to the following table:

{	↤
}	↦
`	↗
|	↕
~	↑

The driver correctly interprets carriage returns, backspaces, tabs, and form-feeds. A new-line that extends over the end of a page is turned into a form-feed. The default line length is 80 characters; indent is 4 characters, and lines per page is 66. Lines longer than the line length minus the indent (i.e. 76 characters, using the above defaults) are truncated.

FILES

/dev/lp

SEE ALSO

lpr(1)

NAME

mem − core memory

DESCRIPTION

Mem is a special file that is an image of the main memory of the computer. 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.

FILES

/dev/mem
/dev/maus/* (minor devices >= 8)

RESTRICTIONS

A single read or write to this driver from the user may not request more than 8128 bytes.

BUGS

Reading minor device 0 will not return an EOF at end-of-file, but will return ENXIO instead.

NAME

mt? – TE16/TU16 magnetic tape interface

DESCRIPTION

The files **mt0**, ..., **mt15** refer to the Digital Equipment Corporation TU16 magnetic tape control and transports. The files **mt0**, ..., **mt7** are 800bpi, and the files **mt8**, ..., **mt15** are 1600bpi. The files **mt0**, ..., **mt3**, **mt8**, ..., **mt11** are designated normal-rewind on close, and the files **mt4**, ..., **mt7**, **mt12**, ..., **mt15** are no-rewind on close. When opened for reading or writing, the tape is assumed to be positioned as desired. When a file is closed, a double end-of-file (double tape mark) is written if the file was opened for writing. If the file was normal-rewind, the tape is rewound. If it is no-rewind and the file was open for writing, the tape is positioned before the second EOF just written. If the file was no-rewind and opened read-only, the tape is positioned after the EOF following the data just read. Once opened, reading is restricted to between the position when opened and the next EOF or the last write. The EOF is returned as a zero-length read. By judiciously choosing *mt* files, it is possible to read and write multi-file tapes.

A standard tape consists of several 512 byte records terminated by an EOF. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time (although very inadvisable).

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the 'raw' interface is appropriate. The associated files are named **rmt0**, ..., **rmt15**. Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, up to the buffer size specified. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. An EOF is returned as a zero-length read, with the tape positioned after the EOF, so that the next read will return the next record.

While doing raw I/O, an EOT will cause a read or write to return error code ENOSPC, indicating that there is no space left on the device.

FILES

/dev/mt
/dev/rmt*

BUGS

If any non-data error (ie. EOT) is encountered while doing block I/O, the driver refuses to do anything more until closed. The driver is limited to four transports.

SEE ALSO

mtm(1)

NAME

nc − network control

DESCRIPTION

The network control pseudo-device provides a means by which a privileged user program can install, remove, and get the status of a BX.25 Permanent Virtual Circuit (PVC) and start, stop, and get the status of a BX.25 link. Additional functions are planned for this driver when the virtual call feature and additional layers of BX.25 are added to the UNIX BX.25 implementation. This driver supports *open, close,* and *ioctl.*

The network control *ioctl* system call has the following form:

ioctl (*fildes*, *cmd*, *arg*)

where *fildes* is the file descriptor returned by the *open* of the nc device and *cmd* is one of the following constants (defined in **/usr/include/sys/nc.h**):

NCPVCI — Install a PVC. This command creates one end of a PVC by connecting a minor device of the X25 driver (slot) to a particular logical channel on a specified link. *Arg* is a pointer to a structure defined as follows:

```
struct pvc {
        unsigned short   slot;
        unsigned short   chno;
        unsigned short   link;
        unsigned short   options;
}
```

where *slot* is the minor device number of the slot to be used as the end point of the PVC, *chno* is the logical channel number to be used, and *link* is the number of the BX.25 link to be used. Links are numbered starding with 0. *Chno* must be in the range 1 to 4,095 and must not be in use currently on the link. The low-order two bits of *options* specify one of three possible session-establishment protocols:

PVC_SESS	session-layer open/close protocol
PVC_RST	reset in-order/out-of-order protocol
PVC_NONE	"no-protocol" session mode

The constants PVC_SESS, PVC_RST, and PVC_NONE are defined in **/usr/include/sys/x25u.h.**

If the link on which the PVC is installed is currently active (not in the halted state), the reset procedure will be initiated for the logical channel. When the reset procedure is completed, the PVC is ready for data transfer.

NCPVCR — Remove a PVC. If *arg* is the minor device number of a slot that is currently associated with a PVC and not open, the local end of that PVC is removed, i.e., disconnected. The slot and logical channel number become available for reuse.

NCSTART — Start a link. *Arg* is a pointer to a structure defined as follows:

```
struct start {
        unsigned short   link;
        unsigned short   vpb;
        unsigned short   kmc;
        unsigned short   line;
```

```
        unsigned short   options;
        unsigned char    cmd[4];
}
```

where *link* is the number of a BX.25 link, *vpb* specifies a minor device number of the VPM interface driver, and *kmc* specifies a minor device number of the KMC driver. If a KMS is being used, *line* specifies which one of the eight synchronous lines (0 through 7) of the KMS that is to be used. If a single-line synchronous interface is being used, this argument must be zero. The four bytes of *cmd* are passed to the protocol script via the *vpmcmd* function.

BX.25 link to a VPM interface driver minor device, and the VPM interface driver minor device to a synchronous line on the KMC. The restart procedure is then initiated for the link.

NCSTOP — Stop a BX.25 link. This command stops the link specified by *arg*. The link data structure is initialized. The link, the associated VPM interface driver, and the KMC synchronous line become available for reuse. While in the halted state packets received for this link are discarded.

NCPVCSTAT — Get the status of a PVC. This command gets the connections and status information for slot *slot* and places it in the *pvc* data structure pointed to by *arg*.

NCLNKSTAT — Get the status of a link. This command gets the connections and status information for link *link* and places it in the *start* data structure pointed to by *arg*.

SEE ALSO

x25pvc(1C), vpm(4), x25(4).

NAME

null — the null file

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return EOF.

FILES

/dev/null

NAME

pcs − program counter sampling device

DESCRIPTION

Pcs provides an interface to program counter sampling, allowing for a statistical approach to user and kernel process profiling. *Pcs* is a read-only pseudo device supporting open, read, close, and ioctl functions. An open of *pcs* obtains exclusive use of the profiling device and starts the profiling clock. The profiling clock is assumed to be a TCU-100 (battery clock) or a KW11-K, if no TCU-100 clock exists. The clock should run at hardware and software priority 7. Thereafter, it is necessary to do an ioctl function to start gathering data. *Pcs* supports the following ioctl requests:

ioctl(fd, PROF_KERNEL, NULL)
: requests that sample points for the unix kernel be output. The third argument is unused.

ioctl(fd, PROF_ALL, NULL)
: requests that sample points be generated whenever any user or kernel process is interrupted. Again, the third argument is unused.

ioctl(fd, PROF_LIST, list)
: requests that sample points for the user level processes specified by the list argument be output. The list argument points to an array of integers; the first element is the number of processes to be sampled, and the remaining values are their process ids. The number of processes in the list is bounded by the number MAX_LIST in **pcs.h**.

ioctl(fd, PROF_GROUP, groupid)
: requests that sample points be output for all of the processes in process group *groupid*.

ioctl(fd, BUF_INCR, incr)
: requests that *incr* more system buffers be allocated for the collection of data. By default, DEF_BUFNO buffers are assigned to the device. *Incr* must be positive, and small enough so that no more than MAX_BUFS are be allocated to the device. (See **pcs.h** for default and max values).

Reads from *pcs* may be for an arbitrary number of bytes, although, in general, partial buffers are not made available to the user until filled with sample points. *Pcs* internally works in terms of standard UNIX buffers, BSIZE bytes in size. Such a buffer is described by the "LOGBUF" structure in **pcs.h**. It is a general layout, envisioned as being useful in reporting kernel and user generated "records" as well as miscellaneous and idle records. Basically, such a buffer consists of a header and several data records. A record is constrained to be no larger than one buffer (minus a buffer header), and in fact, a record is never split across buffers. For this reason, one entry in the header identifies the number of "unused" bytes at the end of the buffer. The unused count should be a small number (less than 12) for all blocks. Anticipating that the operating system will be able to generate records faster than a user process will be able consume them, the header also identifies the number of records "lost" since the last buffer was sent. For streams whose records are generated more slowly than the reading process' ability to consume them, this count should be 0. For high volume data, e.g. pc sampling at a very fast rate, or recording all type of hits on a busy system, this count may be non-zero, and although the data is lost, a count of lost data is provided. To further reduce the possibility of losing data, all system idle counts are stored internally and output every 100 clock cycles. The idle data is identified by sample type IDLE. The same technique is used to gather unasked-for kernel and user data. This data is stored internally and also output every 100 clock cycles as MKERNEL, MKERNELI or MUSER type of sample. If the profiling clock interrupted the processor when it was servicing an interrupt, this data will be output as KERNELI data or it may be stored

internally and output every 100 clock cycles as **MKERNELI** samples.

At the end of sampling, any data that remains in a system buffer is thrown away. This means that as many as three buffers worth of data may be lost when the *close* routine is called.

The data records generated by *pcs* are defined by the structure **PSAMPLE** or **MSAMPLE** in **pcs.h**. For **PSAMPLE** data, the *pid* field gives the process id of the interrupted process, and *pc* the value of its program counter. Also given is the type of sample, kernel or user, and the text space, meaningful only for kernel samples. The cpu interrupt priority level is included in the high 4 bits of *sspace* which again is only meaningful for kernel samples. **MSAMPLE** records are **MKERNEL, MKERNELI, MUSER** and **IDLE** data, *type* identifies the type of data in the sample and *count* is the actual number of hits that were recorded for this type of sample. The **pcs.h** header is as follows:

```
/*              @(#)pcs.h       3.1             */
/*
 *              These structures and macros are used by the SYSTEM PROFILING (pc)
 *              special character device, the data-gathering command getpc,
 *              and the analysis commands analpc.
 *
 *              The pc driver never generates records of types START, STOP, or IOCTL;
 *              these are created by getpc for the benefit of the analpc routines.
 *              The data stream presented by getpc will contain a START record, an
 *              IOCTL record, an arbitrary number of KERNEL, USER, MKERNEL, and
 *              MUSR records, and, finally, a STOP record.
 *              The IOCTL record is an indication of the ioctl system call made by
 *              the getpc program to the pc driver, indicating what data is available
 *              to analpc for reporting.
 */

/*              pc stream record types */

#define KERNEL          1
#define USER            2
#define IDLE            3
#define MKERNEL         4
#define MUSER           5
#define START           6
#define STOP            7
#define IOCTL           8
#define KERNELI         9
#define MKERNELI        10

#define DEF_BUFNO       3               /* default number of system buffers used */
#define MAX_BUFS        10              /* max buffers allowed to pc for profiling */
#define MAX_LIST        5               /* max number in process list for profiling */

struct PSAMPLE {                /* pc profile sample record */
        char            type;           /* KERNEL, KERNELI, or USER space */
        char            sspace;         /* Kernel switchable space number */
                                        /* high 4 bits have CPU priority level */
        short           pid;            /* pid of current user process */
        unsigned pc;    /* program counter */
};

struct MSAMPLE {                /* misc. or merged profile sample record */
        char            type;           /* MKERNEL, MKERNELI, MUSER, or IDLE hit count */
        char            cfill;          /* (structure pad) */
        short           count;          /* number of type of misc. records merged since
                                           last such count */
        unsigned ufill; /* (structure pad) */
};
```

```
struct SSAMPLE {                /* start or stop getpc-produced record */
        char            type;           /* START or STOP */
        char            cfill;          /* (structure pad) */
        time_t          stime;          /* start or stop time */
};

struct ISAMPLE {                /* ioctl record produced by getpc */
        char            type;           /* IOCTL */
        char            cfill;          /* (structure pad) */
        int             cmd;            /* pc ioctl call command */
        short           data[MAX_LIST];         /* ioctl arg., depends on cmd */
};

/*              pc ioctl commands */
#define BUF_INCR            (('P'<<8)|01)   /* incr. number of bufs for /dev/pc */
#define PROF_KERNEL         (('P'<<8)|02)   /* profile the kernel */
#define PROF_ALL            (('P'<<8)|04)   /* profile all user processes */
#define PROF_GROUP          (('P'<<8)|010)  /* profile a user group */
#define PROF_LIST           (('P'<<8)|020)  /* profile a small list of processes */
#define PROF_MASK           0177                    /* used to mask out high byte */

#define WAIT                1
#define NOWAIT              2

#define SAMPPRI             (PZERO+1)

#define NO_CLOCK            0
#define TCU100_CLOCK        1
#define KW11K_CLOCK         2

#define FILLING             1
#define NOT_FILLING         2

struct LOGBUF_HDR {                             /* pc buffer header info */
        ushort          h_unused;       /* num unused bytes at end of buf */
        short           h_numlost;      /* num samples lost between buffers */
};

#define NUMSAMPS   ((BSIZE-sizeof(struct LOGBUF_HDR))/sizeof(struct PSAMPLE))

#define NUMWASTE   (BSIZE-sizeof(struct LOGBUF_HDR)-(NUMSAMPS*sizeof(struct
                                        PSAMPLE)))

struct LOGBUF {                                 /* layout of pc system buffer */
        struct          LOGBUF_HDR lb_buf_hdr;      /* pc buffer header counts */
        struct          PSAMPLE lb_data[NUMSAMPS]; /* pc data samples */
        char            lb_wasted[NUMWASTE];        /* buffer bytes wasted */
};

struct samp_cntl {                              /* pc queue and buffer control info */
        ushort          flag;                           /* control flag */
        struct  buf *cursbuf;           /* current output buffer for sample data */
        caddr_t         currptr;        /* address in block for next sample */
        struct          buf *rq;        /* ready queue (filled for user to read) */
        struct          buf *fq;        /* free queue of empty buffers */
        struct          LOGBUF_HDR sbuf_hdr; /* unused buffer bytes; lost recs count */
};

struct pid_list {                               /* pids of processes to be profiled */
        short p_count;                  /* number in list */
        short p_list[MAX_LIST];         /* list of pids for profiling */
};
```

```
struct DEVPC {                                  /* pc pseudo-device pseudo-registers */
        short           d_clock;                /* type of profiling clock */
        short           d_runpid;               /* pid of user running pc device;
                                                        this is 0 if no one profiling */
        ushort          d_flag;                                 /* type of profiling being done */
        short           d_group;                /* group id profiling for */
        struct pid_list d_list; /* list of pids for profiling */
        struct samp_cntl d_smple; /* queue and buffer control info */
};
```

FILES

/dev/pcs

SEE ALSO

getpc(1), pcstat(1)

NAME

pipe – named pipe

DESCRIPTION

The files identified under **/dev/pipe/*** are called named *pipes*, and are used for one way communications between processes. When a named *pipe* is written, using the file descriptor returned from the open, up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor returned from the open will remove data from the *pipe*. Reading an empty *pipe* will put the process to sleep until there is data available. Since a write to a *pipe* is guarenteed to be atomic, several processes may write a pipe simultaneously without their individual writes intermixing.

As long as at least one process has either the reading or the writing end of a pipe open, any data that is in the pipe is preserved. When the last reference to a pipe is gone (closed), any data that is in the pipe is discarded.

Ioctl (2) can be used to cause the process not to sleep when an empty or full pipe is encountered. It is used as follows:

```
#include <sys/ioctl.h>
ioctl(fd, FIOSPIPE, &addr);    /* used to set the mode */
ioctl(fd, FIOGPIPE, &addr);    /* used to get the mode */
```

Addr is a two byte structure; the first byte is the read flag and the second byte is the write flag. The flag set to 0 means do not sleep on a write to a full pipe or a read of an empty pipe. This causes a 0 to be returned from the respective system call. A 1 in the flag indicates the process will sleep on the above conditions.

FILES

/dev/pipe/*

SEE ALSO

ioctl(2)

NAME

rk? – RK11/RK03 or RK05 disk

DESCRIPTION

Rk? refers to an entire RK03 disk as a single sequentially-addressed file. Its 256-word blocks are numbered 0 to 4871.

The *rk* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single *read* or *write* call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw *RK* files begin with **rrk** and end with a number which selects the same disk as the corresponding **rk** file.

In raw I/O, the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise, calls to *lseek* (2) should specify a multiple of 512 bytes.

FILES

/dev/rk*
/dev/rrk*

NAME

rootdev – root file system

DESCRIPTION

The root file system is the heart of all UNIX activity. Its size and location on the disk are determined when UNIX is compiled. The root file system must be present at boot time since all files are referenced in some way from the root file system. When **unix.util** is booted, the root file system is redefined to be in util (the very first few cylinders of the disk).

FILES

/dev/rootdev

SEE ALSO

hp(4)

NAME

rx? — floppy disk

DESCRIPTION

The floppy disk is an easily dismountable block storage device that will hold 500 blocks of data or 1000 blocks, if dual density. In single density mode, there is an extra 256 byte half block. This is because there are 2002 sectors of data on every floppy disk and 128 bytes of data in each sector of a single density floppy. The current devices are:

BLOCK	CHARACTER	DEVICE
/dev/rx0	/dev/rrx0	rx01
/dev/rx1	/dev/rrx1	rx01
/dev/rx2	/dev/rrx2	rx02
/dev/rx3	/dev/rrx3	rx02

The block interface is useful for installing file systems. The character interface must be used to read or set the density of the current diskette. The variable density feature is only available on the **rx02**.

The command *flopden*(1) is useful for manipulating the density of a diskette. To either read the current density or change the density from an application program, the character device interface is opened, and an *ioctl*(2) system call is made. (See **/usr/include/sys/rx.h** for the correct define symbols.)

The driver has only one queue, and thus simultaneous reads on multiple controllers are not possible. However it is not limited to the number of controllers; the only action necessary to add another controller is a simple addition of another device address in the list of device addresses within the source of the driver. The driver dynamically determines whether the controller is an **rx01** or **rx02**.

FILES

/dev/rx?
/dev/rrx?
/usr/include/sys/rx.h

SEE ALSO

flopden(1), ioctl(2)

NAME

swapdev – location for swapping

DESCRIPTION

The swap area of the disk is used by UNIX when it has to remove an active process from core to make room for others. The swap area is initialized at boot up. Its size and location on the disk are determined when UNIX is compiled. When **unix.util** is booted, the swap area may or may not be redefined

FILES

/dev/swapdev

SEE ALSO

hp(4)

NAME

tm? − TM11/TU10 magnetic tape interface

DESCRIPTION

The files **tm0**, ..., **tm7** refer to the Digital Equipment Corporation TM11/TU10 magnetic tape control and transports at 800bpi. The files **tm0**, ..., **tm3** are designated normal-rewind on close, and the files **tm4**, ..., **tm7** are no-rewind on close. When opened for reading or writing, the tape is assumed to be positioned as desired. When a file is closed, a double end-of-file (double tape mark) is written if the file was opened for writing. If the file was normal-rewind, the tape is rewound. If it is no-rewind and the file was open for writing, the tape is positioned before the second EOF just written. If the file was no-rewind and opened read-only, the tape is positioned after the EOF following the data just read. Once opened, reading is restricted to between the position when opened and the next EOF or the last write. The EOF is returned as a zero-length read. By judiciously choosing *tm* files, it is possible to read and write multi-file tapes.

A standard tape consists of several 512 byte records terminated by an EOF. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time (although very inadvisable).

The *tm* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the 'raw' interface is appropriate. The associated files are named **rtm0**, ..., **rtm7** Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, up to the buffer size specified. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. An EOF is returned as a zero-length read, with the tape positioned after the EOF, so that the next read will return the next record.

While doing raw I/O an EOT will return a error indicating that there is no space left on the device.

FILES

/dev/mt*
/dev/rtm*

BUGS

If any non-data error (ie. EOT) is encountered while doing block I/O it refuses to do anything more until closed: The driver is limited to four transports.

SEE ALSO

mtm(1)

NAME

trace − event-tracing driver

DESCRIPTION

Trace is a special file that allows event records generated within the UNIX kernel to be passed to a user program so that the activity of a driver or other system routines can be monitored for debugging purposes.

An event record is generated from within a kernel driver or system routine by invoking the *trsave* function:

```
trsave(dev, chno, buf, cnt)
char   dev, chno, *buf, cnt;
```

Dev is a minor device number of the trace driver; *chno* is an integer between 0 and 15 inclusive that identifies the data stream (channel) to which the record belongs; *buf* is a buffer containing the bytes that make up a single event record, and *cnt* is the number of bytes in *buf*. Calls to *trsave* will result in data being placed on a queue, provided that some user program has opened the trace minor device *dev* and has enabled channel *chno*. Event records prefaced by *chno* and *cnt* are stored on a queue until a system-defined maximum (TRQMAX) is reached; an event record is discarded if there is not sufficient room on the queue for the entire record. This implies that event records with *cnt* > TRQMAX − 2 are discarded. The queue is emptied by a user program reading the trace driver. Each *read* returns an integral number of event records; the read count must, therefore, be at least equal to the size of a record plus two.

The *trace* driver supports *open*, *close*, *read*, and *ioctl* system calls. The *ioctl* system call is invoked as follows:

```
#include <sys/vpm.h>
int fildes, cmd, arg;
ioctl(fildes, cmd, arg);
```

The *trace ioctl* commands are:

VPMSETC Enable trace channels. This command enables the channels indicated by a 1 in the bit mask found in *arg*. The low-order bit (bit 0) corresponds to channel zero, the next bit (bit 1) corresponds to channel 1, etc..

VPMGETC Get enabled channels. This command returns in *arg* a bit mask containing a 1 for each channel that is currently enabled.

VPMCLRC Disable channels. This command disables the channels indicated by a 1 in the bit mask found in *arg*.

SEE ALSO

vpmstart(1C), vpm(4).

NAME

tty – general interface for terminals

DESCRIPTION

This section describes both a particular special file and the general nature of the terminal interface.

The file **/dev/ln** is, in each process, a synonym for the control terminal associated with that process. 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.

As for terminals in general: all of the low-speed asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of the interface; *dh* (4) and *kl* (4) describe peculiarities of the individual devices.

When a terminal file is opened, it causes a wait to take place at the first read or write. In practice, user's programs seldom open these files; they are opened by *init* (1M) and become a user's input and output files. The very first terminal file open in a process becomes the *control terminal* for that process. 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 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 choked, 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.

These special files have a number of modes that can be changed by use of the *ioctl* (2) system call. When first opened, the interface mode is 300 baud, either parity accepted, and 10 bits/character (one stop bit). Subsequent opens do not change the modes or speeds even if all the processes referencing the line have closed the line. Modes that can be changed by *ioctl* include the interface speed (if the hardware permits); number of data and stop bits; acceptance of even parity, odd parity, or both; a raw mode in which all characters may be read, and all 8 bits are sent on output (see *ioctl* (2)); a carriage return (CR) mode in which CR is mapped into new-line on input and in which both CR and line-feed (LF) cause the echoing of the sequence CR-LF; mapping of upper-case letters into lower case; suppression of echoing; a variety of delays after function characters; and the printing of tabs as spaces.

Normally, terminal input is processed in units of lines. 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 # erases the last character typed, except that it will not erase beyond the beginning of a line or an ASCII EOT. By default, the character @ kills the entire line up to the point where it was typed, but not beyond an EOT, and causes a carriage return. Both these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. Both @ and # may be entered literally by preceding them with a \; the @ and # remain, but the \ disappears. These erase and kill characters may be changed.

When desired, all upper-case letters are mapped into the corresponding lower-case letters. In this made, an upper-case letter may be generated by preceding it with \. In addition, the

following escape sequences are generated on output and accepted on input:

for:	*use:*
`	\'
|	\!
~	\^
{	\(
}	\)

In *raw* mode, no erase or kill processing is done for the reading program; and the EOT, quit, and interrupt characters are not treated specially. Control is returned to the reading program only when the *read* (2) character count has been satisfied (as well as if an *alarm* (2) signal occurs, or if the line hangs up). The input parity bit is passed back to the reader. On output, all 8-bits are sent if parity is set to *even* and *odd* and the number of data bits is set to 8.

The ASCII EOT (control-d) character may be used to generate an end-of-file from a terminal. When an EOT is received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOT is discarded. Thus, if there are no characters waiting, which is to say the EOT occurred at the beginning of a line, zero characters will be passed back, and this is the standard end-of-file indication. The EOT is passed back unchanged in raw mode.

The ASCII DC3 (control-s) character can be used to temporarily stop output. It is useful with CRT terminals to prevent output from disappearing before it can be read. Output is resumed when the ASCII DC1 (control-q) character is typed. These start/stop characters are not passed to any other program when used in this manner. Output may also be stopped by typing an ESC character. In this case output is resumed by typing any character. A BREAK character is treated like an ESC character when not in raw mode.

When the carrier signal from the data-set drops (usually because the user has hung up his terminal), 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 read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file on their input can terminate appropriately when hung up on.

Two characters have a special meaning when typed. The ASCII DEL character (sometimes called "rubout") is not passed to a program, but generates an *interrupt* signal that is sent to all processes associated with the 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).

The ASCII FS character generates the *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 generated in the working directory.

When one or more characters are written, they are actually 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. When 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.

FILES

/dev/ln*

SEE ALSO

ioctl(2), signal(2), dh(4), kl(4).

NAME

vp – Versatec printer-plotter

DESCRIPTION

Vp0 is the interface to a Versatec D1200A printer-plotter with a Versatec C-PDP11(DMA) controller. Ordinarily bytes written on it are interpreted as ASCII characters and printed. As a printer, it writes 64 lines of 132 characters each on 11 by 8.5 inch paper. Only some of the ASCII control characters are interpreted.

NL performs the usual new-line function, i.e. spaces up the paper and resets to the left margin. It is ignored however following a CR which ends a non-empty line.

CR is ignored if the current line is empty but is otherwise like NL.

FF resets to the left margin and then to the top of the next page.

EOT resets to the left margin, advances 8 inches, and then performs a FF.

The *ioctl* (2) system call may be used to change the mode of the device as follows:

```
#include <sys/ioctl.h>
short   cmd = 0XXX;
        :
        :
        :
ioctl(fd, VIOCSETD, &cmd);  /* used to set the mode */
```

or

```
ioctl(fd, VIOCGETD, &cmd);  /* used to get the mode */
```

The bits (XXX) mean:

0400 Enter simultaneous print/plot mode.
0200 Enter plot mode.
0100 Enter print mode (default on open).
040 Send remote terminate.
020 Send remote form-feed.
010 Send remote EOT.
04 Send remote clear.
02 Send remote reset.

When opened, a reset, clear, and form-feed are performed automatically. Notice that the mode bits are not encoded, so that it is required that exactly one be set.

In plot mode each byte is interpreted as 8 bits of which the high-order is plotted to the left; a '1' leaves a visible dot. A full line of dots is produced by 264 bytes; lines are terminated only by count or by a remote terminate function. There are 200 dots per inch both vertically and horizontally.

When simultaneous print-plot mode is entered exactly one line of characters, terminated by NL, CR, or the remote terminate function, should be written. Then the device enters plot mode and at least 20 lines of plotting bytes should be sent. As the line of characters (which is 20 dots high) is printed, the plotting bytes overlay the characters. Notice that it is impossible to print characters on baselines that differ by fewer than 20 dot-lines.

In print, mode lines may be terminated either with an appropriate ASCII character or by using the remote terminate function.

FILES

/dev/vp0

NAME

vpm, vpb — Virtual Protocol Machine Protocol and Interface Drivers

DESCRIPTION

This entry describes the *vpm* and *vpb* drivers and gives an introduction to the Virtual Protocol Machine (VPM).

VPM is a software package for implementing link-level protocols on the DEC KMC11 microcomputer in a high-level-language. This is accomplished by a compiler that runs on UNIX and translates a high-level language description of a protocol into an intermediate language that is executed by an interpreter running in the KMC. VPM also provides a framework for implementing higher levels of protocols (levels 3 and above) as UNIX drivers.

The VPM software consists of the following components:

1. A compiler (*vpmc*(1C)) for the protocol description language; it runs on UNIX.
2. An interpreter that controls the overall operation of the KMC and interprets the protocol script.
3. Two UNIX drivers: A Protocol driver and an Interface driver.
4. *vpmstart*(1C): a UNIX command that copies a load module into the KMC and starts it.
5. *vpmset*(1C): a UNIX command that logically connects VPM minor devices and KMC synchronous links. *x25pvc*(1C) and *x25lnk*(1C) may also be used to connect things up.
6. *vpmsnap*(1C): a UNIX command that prints a time-stamped event trace while the protocol is running.
7. *vpmtrace*(1C): a UNIX command that prints an event trace for debugging purposes while the protocol is running.
8. *vpmsave*(1C): a UNIX command that writes unformated trace data to its standard output.
9. *vpmfmt*(1C): a UNIX command that formats the output of *vpmsave*.

The VPM protocol driver provides a simple user interface to a synchronous line controlled by a link-level protocol executed by the VPM interpreter in the KMC. It supports the following UNIX system calls: *open*, *read*, *write*, *close*, and *ioctl*. If higher levels of protocol are required, the VPM protocol driver may be modified or replaced. The VPM interface driver provides a common interface to a synchronous line controlled by a link-level protocol executed by the VPM interpreter. This common interface can be shared by several different protocol modules (see *x25*(4)).

Before a protocol driver minor device can be used, it must be logically connected to a VPM interface driver; the interface driver minor device must in turn be logically connected to a synchronous line of a KMC microprocesser or a KMS11 communication multiplexor. These corrections can be made by means of *ioctl* commands (see below). The command *vpmset*(1C) uses these *ioctl* commands to make these connections.

The VPM Interface Driver.

The VPM interface driver provides a general purpose interface between level 3 protocols executing in the UNIX kernel and level 2 protocols being executed by the VPM interpreter in the KMC. This interface is used by the VPM Protocol driver as well other protocol drivers such as the LEAP, X25, and ST.

The Interface Driver supports *open, close,* and *ioctl* systems calls. These calls are used to set-up connections between the interface driver and a synchronous line on a KMC or KMS and to set interpreter options. The system *ioctl* call has the following form:

ioctl (fildes, cmd, arg)

Possible values for the *cmd* argument are:

VPMSDEV — Connect an interface driver minor device to a synchronous line on a KMC or KMS. Bits 6 and 7 of *arg* contain the minor device number of the KMC or KMS. Bits 0-2 of *arg* contain the line number (0-7) if a KMS is being used; for a single-line KMC they must be zero.

VPMGETM — Get the interpreter modes. The currently available modes are the normal mode and the X.25 mode. The normal mode is indicated by an *arg* of all zeros. In this mode, the entire information field of an I frame is copied by the interpreter to the assigned buffer. The X.25 mode is indicated by a 1 in bit 0. In this mode, the VPM interpreter copies the first three bytes of the information field of level 2 I frames into the buffer descriptor of the buffer assigned to receive the frame. These three bytes are the X.25 level 3 header. The remaining bytes, if any, are copied to the buffer pointed to by the buffer descriptor.

VPMSETM — Set the interpreter modes. The modes specifed by a 1 in the mask *arg* are set.

VPMCLRM — Clear interpreter modes. The modes specified by a 1 in the mask *arg* are cleared.

The routines that make up the VPM interface are:

vpmstart — Start the level 2 protocol.

vpmstop — Stop the level 2 protocol.

vpmxmtq — Place a transmit buffer descriptor pointer on the level 2 transmit queue.

vpmempty — Place a empty receive buffer descriptor pointer on the level 2 empty receive queue.

vpmcmd — Send a four byte command to the level 2 protocol.

vpmrpt — Receive a four byte report from the level 2 protocol.

vpmenq — Place a buffer descriptor pointer at the end of the indicator VPM linked list queue.

vpmdeq — Remove the buffer descriptor at the head of the indicated VPM linked list queue.

vpmrmv — Search the indicated VPM linked list queue for the given buffer descriptor pointer and remove it if found.

vpmerr — Get the error counters maintain by the VPM interpreter. After the interpreter has passed the counters to the driver it resets its copy of the counters.

vpmsave — Save an event record using the trace driver minor device zero.

vpmsnap — Save a time-stamped event record using the trace driver minor device 1.

Operation of the Standard Protocol Driver.

UNIX user processes transfer data to or from a remote terminal or computer system through VPM using normal *open*, *read*, *write*, and *close* operations. Flow control and error recovery are provided by the protocol executed by the interpreter in the KMC.

The VPM *open* for reading-and-writing is exclusive; *open*s for reading-only or writing-only are not exclusive. The VPM *open* checks that the correct interpreter is running in the KMC and then sends a command to the interpreter which causes it to start interpreting the protocol script. The driver then supplies one or more 512-byte receive buffers to the interpreter.

The VPM *read* returns either the number of bytes requested or the number remaining in the current receive buffer, whichever is less; any bytes remaining in the current receive buffer are used to satisfy subsequent reads. The VPM *write* copies the user data into 512-byte system buffers and passes them to the VPM interpreter in the KMC for transmission.

The VPM *close* arranges for the return of system buffers and for a general cleanup when the last transmit buffer has been returned by the interpreter. It also stops the execution of the protocol script.

The VPM protocol driver supports the following *ioctl* commands:

VPMCMD — Send a command to the protocol script. The first four bytes of the array pointed to by *arg* are passed to the VPM interpreter which saves them and passes them to the protocol script when it executes a *getcmd* primitive. Only the most recent command is kept by the VPM interpreter.

VPMERRS — Get and then reset the interpreter's error counters. The interpreter's four, two-bytes error counters are copied to the array pointed to by *arg*. The interpreter's copy of the counters is then set to zero.

VPMRPT — Get the latest script report. When the protocol script executes a *rtnrpt* primitive, a four-byte report is passed from the protocol script to the VPM protocol driver. Only the most recent script report is kept by the driver. If there is a script report that has not previously been passed to a user via this *ioctl* command, that report is copied to the array pointed to by *arg* and a non-zero value (*one*) is passed as the return value. If no script report is available, a *zero* is passed as the return value.

VPMSDEV — Connect a protocol driver to an interface driver. *Arg* is the minor device number of the interface driver to be connected to this protocol driver. To invoke this *ioctl* command, the file status flag, O_NDELAY must be set.

The VPM Event Trace

The VPM drivers generates a number of event records to allow the activity of the drivers and protocol script to be monitored for debugging purposes. If a program such as *vpmtrace*(1C) or *vpmsave*(1C) has opened minor device 0 of the trace driver and has enabled the appropriate channels on that device, these event records are queued for reading; otherwise, the event records are discarded by the trace driver. Event records associated with interface driver minor device n are put on the read queue for minor device 0 of the trace driver with a channel number of n. Calls to the system functions *vpmopen*, *vpmread*, *vpmwrite*, and *vpmclose* generate event records identified respectively by **o**, **r**, **w**, and **c**. Calls to the *vpmc*(1C) primitive *trace*(*arg1*,*arg2*) cause the VPM interpreter to pass *arg1* and *arg2* along with the current value of the script location counter to the VPM driver, which generates an event record identified by a **T**. Each event record is structured as follows:

```
struct event {
        short   e_seqn;         /* sequence number */
        char    e_type;         /* record identifier */
        char    e_dev;          /* minor device number */
        short   e_short1;       /* data */
        short   e_short2;       /* data */
}
```

When the script terminates for any reason, the driver is notified and generates an event record identified by an **E**. This record also contains the minor device number, the script location counter, and a termination code defined as follows:

0 Normal termination; the interpreter received a HALT command from the driver.
1 Undefined virtual-machine operation code.
2 Script program counter out of bounds.
3 Interpreter stack overflow or underflow.
4 Jump address not even.
5 UNIBUS error.
6 Transmit buffer has an odd address; the driver tried to give the interpreter too many transmit buffers; or a *get* or *rtnxbuf* was executed while no transmit buffer was open, i.e., no *getxbuf* was executed prior to the *get* or *rtnxbuf*.
7 Receive buffer has an odd address; the driver tried to give the interpreter too many receive buffers; or a *put* or *rtnrbuf* was executed while no receive buffer was open, i.e., no *getrbuf* was executed prior to the *get* or *rtnxbuf*.
8 The script executed an *exit* primitive.
9 A *crc16* was executed without a preceding *crcloc* execution.
10 The interpreter detected loss of the modem-ready signal at the modem interface.
11 Transmit-buffer sequence-number error.
12 Command error: an invalid command or an improper sequence of commands was received from the driver.
13 Not used.
14 Invalid transmit state (internal error).
15 Invalid receive state (internal error).
16 Not used.
17 *Xmtctl* or *setctl* attempted while transmitter was still busy.
18 Not used.
19 Same as error code 6.
20 Same as error code 7.
21 Script too large.
22 Used for debugging the interpreter.
23 The driver's OK-check has timed out.

SEE ALSO

vpmc(1C), vpmset(1C), vpmstart(1C), x25lnk(1C), x25pvc(1C), trace(4).

NAME

vt − graphics interface

DESCRIPTION

OUTPUT: **/dev/vt** (crt − only one user)

Data may be displayed on a vt11 graphics tube by a write system call:

write (fdo, out_buf, count);

where *fdo* is an integer file descriptor, *out_buf* is an integer array containing the display list and *count* is an integer containing the number of bytes in the display list and must be even. The display list is a sequence of octal numbers that define the image to be drawn. (These octal numbers are a mixture of control words and data that are given to the vt11 microprocessor.)

Prior to the write, the device must have been opened by

fdo = open ("/dev/vt11", 1);

and a seek must have been made to the proper frame

lseek (fdo, *n*, **0);**

where *n* is long and indicates the frame number (0 thru 9). A frame is an independently modifiable overlay which when overlayed with other frames complete the image.

The following is an example of a user program that will draw a 0200 by 0200 unit box at location 0500,0500 on the screen:

```
main()
{
        char *file;
        int fd;
        static int buf[]{0117124, 0500, 0500, 0110000, 040200,
                0, 040000, 0200, 060200, 0, 040000, 020200};

        file = "/dev/vt";
        fd = open(file, 1);
        if(fd < 0) {
                printf("failed to open %s0, file);
                exit(0);
        }
        lseek(fd, 0L, 0);
        write(fd, buf, sizeof(buf));
        for(;;)
                sleep(3600);
}
```

INPUT: **/dev/vtlp** (light pen) or **/dev/vtjy** (joy stick)
(only one user each device)

After an open system call: **fdi = open ("/dev/vtlp",0)** or **fdi = open ("/dev/vtjy",0)**) input data can be obtained by a read system call:

in_count = read (fdi, in_buf, count);

where *in_buf* is a 3 element integer array.

If *count* is 0, the process will sleep until input occurs (event 1 or 2).

If *count* is 6, the *read* will return immediately and the 3 integers of *in_buf* contain: **event**, **x**, **y**. Where *x* and *y* are integers and contain the x and y coordinates respectively.

If *event* is 0, there is no unserviced input (event 1 or 2).

If *event* is 1, tracking start or a button is released.

If *event* is 2, tracking is stopped.

SYSTEM: (system proc table)

When a user graphics program is not running, the vtll may be used to display the operating system proc table. A sample of the proc table is shown below:

CB-UNIX Release 2.1 0000:07:46
100 procs 30 texts

s	fl	wchan	sg	pri	ptm	ctm	clock	group	pid	ppid	size	name
s:	u	22656		-100	127	127		0	0	0	20	UNIX Scheduler
s:		24756	0	40	127	127		0	1	0	131	init
s:		25014	0	10	127	3	722	0	4	1	111	su
r:		*0*	*0*	*10*	*3*	*2*		*4*	*10*	*4*	*216*	*ls −l*

FILES

/dev/vt
/dev/vtlp
/dev/vtjy

SEE ALSO

lseek(2), open(2), read(2), write(2)

RESTRICTIONS

Double word vtll instructions must NOT begin at out_buf[i] where i % 254 == 253 or grave disorder will result.

NAME

vtp − virtual terminal protocol

DESCRIPTION

This section describes how to use the virtual terminal protocol feature that can be optionally enabled in the operating system.

The virtual terminal protocol provides a means whereby user programs can be written to interact with CRT terminals in a language which is independent of the actual type of terminal. The operating system provides translation of standard sequences of characters into the sequences necessary to cause the desired behavior on each type of terminal. It also translates what the terminal sends to the system so that the user program sees the standard sequence for all terminals regardless of the terminal actually in use.

An *ioctl*() is necessary to enable a specific terminal handler and that terminal handler must have been compiled into the operating system. The *ioctl* call is described in <**sys/termio.h**>:

```
/*
 * structure of ioctl arg for LDGETT and LDSETT
 */
struct      termcb      {
            char        st_flgs;    /* term flags */
            char        st_termt;   /* term type */
            char        st_crow;    /* gtty only - current row */
            char        st_ccol;    /* gtty only - current col */
            char        st_vrow;    /* variable row */
            char        st_lrow;    /* gtty only - last row */
};
```

Terminals for which drivers are currently available are the DEC vt61 and vt100, the TEC scope, the Teletype D40, the Hewlett Packard hp26xx terminals, and the Concept 100.

The terminal flags are automatically set by the *ioctl*() on a **LDSETT** command to appropriate values for a specific terminal unless the user overrides these defaults by setting the **TM_SET** bit. If the user does this, then the terminal flags are set according to the other flags found in "st_flgs".

TM_SNL The special newline flag means that the newline character will be treated specially. Currently this is used by the Dataspeed 40 terminals. When this flag is set, newline characters are converted to "load cursor address sequences" so that the printed newline character doesn't appear on the screen.

TM_ANL Causes an automatic newline whenever the cursor attempts to pass the 80th column.

TM_LCF The "last column function" flag causes scrolling to be emulated on dumb terminals, such as the TEC, which do not have scrolling hardware. Scrolling is emulated by moving the cursor to the first row of the terminal, deleting the line, and then moving the cursor to the bottom of the screen again.

TM_CECHO

Causes the cursor motion keys to function without user software intervention by causing the codes generated by the cursor control keys to be immediately echoed back to the terminal when they are received. This flag is usually used in conjunction with the **TM_CINVIS** flag.

TM_CINVIS

Inhibits the translation and tranmission of the cursor motion sequences to the user program. If this flag and the **TM_CECHO** flag are on, that the cursor

motion keys work without intervention by the user process.

TM_SET Causes tne values of the preceeding flags to be set or cleared if the **LDSETT** *ioctl*() command is being done.

"st_vrow" specifies the row at which scrolling will take place. This means that everything on the screen above that row will be unaffected as material scrolls upwards from the bottom. This allows split screen operation. "st_crow" and "st_ccol" contain the system's idea of the current row and column when a **LDGETT** command is done. "st_lrow" contains the system's idea of which row is the last row visible on the CRT screen. To assure that the system and the terminal both agree on the cursor position, a **VHOME** escape sequence should be transmitted to the terminal after the terminal handler is enabled. Columns and rows are numbered from (0,0).

Once the terminal type is set, user programs use the escape sequences described in **/usr/include/sys/crtctl.h** to control the behavior of the terminal.

```
/*              @(#)crtctl.h    3.2             */

/*
                Define the cursor control codes
*/
#define ESC     033             /* Escape for command */

/* Commands */
#define CUP     0101            /* Cursor up */
#define CDN     0102            /* Cursor down */
#define CRI     0103            /* Cursor right */
#define CLE     0104            /* Cursor left */
#define HOME    0105            /* Cursor home */
#define VHOME   0106            /* cursor home to variable portion */
#define LCA     0107            /* Load cursor, followed by (x,y) in (col,row) */

#define STB     0110            /* Start blink */
#define SPB     0111            /* Stop blink */
#define CS      0112            /* Clear Screen */
#define EEOL    0113            /* Erase to end of line */
#define EEOP    0114            /* Erase to end of page */
#define DC      0115            /* Delete character */
#define DL      0116            /* Delete line */
#define IC      0117            /* Insert character */
#define IL      0120            /* Insert line */
#define KBL     0121            /* keyboard lock */
#define KBU     0122            /* keyboard unlock */
#define ATAB    0123            /* Set Column of tabs on all lines */
#define STAB    0124            /* Set single tab on current line only */
#define CTAB    0125            /* Clear all tabs */
#define CSTAB   0144            /* Clear tab at current column, all lines */
#define USCRL   0126            /* Scroll up one line */
#define DSCRL   0127            /* Scroll down one line */
#define ASEG    0130            /* Advance segment */
#define BPRT    0131            /* Begin protect */
#define EPRT    0132            /* End  protect */

#define CRTN    0133            /* Return cursor to beginning of line */
#define NL      0134            /* Terminal newline function */
#define CM      0135            /* Clear Memory (Terminal Reset) */
#define SVSCN   0136            /* Define variable portion of screen (OS only) */
#define UVSCN   0137            /* Scroll Up variable portion of screen */
#define DVSCN   0140            /* Scroll Down variable portion of screen */
```

```
#define SVID    0141        /* Set Video Attributes */
#define CVID    0142        /* Clear Video Attributes */
#define DVID    0143        /* Define Video Attributes */
/* Video Attribute Definitions */
#define VID_NORM            000         /* normal */
#define VID_UL              001         /* underline */
#define VID_BLNK            002         /* blink */
#define VID_REV             004         /* reverse video */
#define VID_DIM             010         /* dim intensity */
#define VID_BOLD            020         /* bright intensity */
#define VID_OFF             040         /* blank out field */

#define BRK     000         /* transmit break */
#define HIQ     001         /* Put remainder of this write on the high
                               priority queue, saving current cursor and restoring
                               when done. */
```

When sending escape sequences to the terminal, it is necessary that the writes be atomic. In other words, if it is desired to move the cursor to column 10 and row 5, it is necessary to send the four characters, ESC LCA 10 5, to the terminal in a single write system call. If standard I/O is being used, the stream to the terminal must be buffered, and then flushed after the escape sequence is written so that the write is atomic.

Note that some of these functions will not work with every terminal. Whether a function works or not is dependent on how smart the terminal handler code in the operating system is and what capabilities the terminal itself has. Basically you can expect that all terminal handlers can manage the cursor motion commands, CUP, CDN, CRI, CLE, HOME, VHOME, and LCA. Most terminal handlers can also do the scrolling of the variable portion of the screen, UVSCN and DVSCN, though sometimes the terminal handler emulates scrolling by deleting a line at the top of the region and then writing a new one at the bottom.

If a terminal has advanced video features such as blinking, underlining, and reverse video, it is possible to turn these features on and off with the SVID, CVID, and DVID commands. The "set video attributes", SVID, logically ors in the features specified by the next character. "Clear video attributes", CVID, and complements out the features specified by the next character. "Define video attributes", DVID, replaces the current video attributes with those specified by the next character.

Of particular utility is the "hi-queue write", HIQ. When combined with the variable scroll feature, it is possible to prevent some section at the top of the screen from being changed by normal writes and then have a special program perform hi-queue writes, which contain a "load cursor address" function (LCA) to modify the contents at the top of the screen. This allows the possibility of having a background process keep a display at the top of the screen updated while the user continues working in the lower portion of the screen. Hi-queue writes are limited to 512 bytes.

DEFICIENCIES

One annoying fact is that the HIQ string is terminated by the end of the write system call.

NAME

X25 − BX.25 network interface

DESCRIPTION

The X25 driver provides multiplexed channels over one or more synchronous communications lines using the Bell System standard BX.25 Level 3 protocol. The current release supports permanent virtual circuits (PVCs) only; the call set-up features needed to support virtual calls have not yet been implemented. There is a separate and independent Level 3 interface for each communications line. Point-to-point connections between hosts are supported as well as connections via an X.25 network.

The X25 driver is implemented as a VPM protocol module (see *vpm*(4)). The X25 driver uses the VPM interface module to access communications lines controlled by KMC11-B microprocessors. Level 2 of BX.25, the link level, is implemented by a VPM protocol script in the KMC.

The special files **/dev/x25/s?** refer to the minor devices of the X25 driver. Each such minor device, also referred to as a *slot*, can be connected by means of a *network control* device (see *nc*(4)) to an arbitrary logical channel (1-4095) on a specified X25 interface. When the other end of the logical channel has been connected in an analogous fashion, each slot so connected is the terminus of a *permanent virtual circuit*, which is a full-duplex connection over a BX.25 logical channel between a set of user processes on the local host and another set of user processes on a remote host. A logical channel is a connection which may be multiplexed with other channels over a physical link to a remote host or an X.25 network. Each X25 interface (also referred to as a *link*) must be connected via the network-control device to a particular KMC microprocessor or to a particular line on a KMS11 communications multiplexor.

A user process accesses a BX.25 minor device (slot) using *open*, *close*, *read*, *write*, and *ioctl* system calls.

There are several internal flags that are maintained by the X25 driver for each slot. The values of these flags can be read and in some cases modified by means of the *ioctl* system call (see below).

An *open* will fail and return the error EIO if the specified slot does not exist, if the slot is not currently connected to a logical channel on some link, or if the link to which the slot is connected is not currently active. The user may request the normal *open* options O_RDONLY, O_WRONLY, and O_RDWR. The user may also request that reads with no data available should not sleep, writes with no transmit queue space return immediately, and that *open* should not wait for *faropen* to be set, using the O_NDELAY *open* flag. The *open*, and all use of the slot, can be made exclusive, using the O_EXCL *open* flag. If an exclusive *open* is requested and cannot be granted, the error EBUSY will be returned. A successful *open* will clear the *isreset* status bit (see the discussion of *ioctl* below). If O_NDELAY is specified, the user is responsible for insuring that the remote end of the slot is ready to receive data before any is sent via *writes*. Note that O_NDELAY can be set via the *fcntl*(2) system call after a successful *open* (via the *ioctl* call **X25FCNTL** for CB-UNIX), which insures that the *open* will not return until the other end is fully connected.

An *open* may or may not block until the far end is also open, depending on the session-establishment protocol requested. There are three choices for the session-establishment protocol. The choice is made by means of the network-control device at the time the permanent virtual circuit is installed. The first mode, referred to as the "no-protocol" session mode, is for the *open* to return immediately. This puts the burden on the user program to determine whether the far end is actually open. The reset session mode, designed mainly for compatibility with certain non-UNIX implementations of BX.25, uses a RESET in-order packet to indicate to the far end that a slot has been opened and a RESET out-of-order packet to indicate to the far end that the slot has been closed. In the current implementation, the RESET in-order and RESET out-of-order packets are recognized when they are received, but are not transmitted