Dansk Data Elektronik A/S

SUPERMAX BASIC UTILITIES

System V Reference Manual

Section 4 – 9

Release 3.1, Version 1.0

Stock no.: 94300131

Issued March 1990

NAME

intro – introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C structure declarations for the file formats are given where applicable. Usually, the header files containing these structure declarations can be found in the directories **/usr/include** or **/usr/include/sys**. For inclusion in C language programs, however, the syntax **#include <filename.h>** or **#include <sys/filename.h>** should be used.

This page is intentionally left blank

NAME

a.out – common assembler and link editor output

SYNOPSIS

#include <a.out.h>

DESCRIPTION

The file name **a.out** is the default output file name from the link editor *ld*(1). The link editor will make *a.out* executable if there were no errors in linking. The output file of the assembler *as*(1), also follows the common object file format of the *a.out* file although the default file name is different.

A common object file consists of a file header, an (optional) UNIX system header (if the file is link editor output), a table of section headers, relocation information, (optional) line numbers, a symbol table, and a string table. The order is given below.

File header.
UNIX system header (optional).
Section 1 header.
...
Section n header.
Section 1 data.
...
Section n data.
Section 1 relocation.
...
Section n relocation.
Section 1 line numbers.
...
Section n line numbers.
Symbol table.
String table.

The last three parts of an object file (line numbers, symbol table and string table) may be missing if the program was linked with the **–s** option of *ld*(1) or if they were removed by *strip*(1). Also note that the relocation information will be absent after linking unless the **−r** option of *ld*(1) was used. The string table exists only if the symbol table contains symbols with names longer than eight characters.

The sizes of each section (contained in the header, discussed below) are in bytes.

When an **a.out** file is loaded into memory for execution, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), and a stack. On the Supermax computer the text segment starts at location 0x200000.

The data segment starts at the next megabyte boundary past the last text address.

On a Supermax computer equipped with MC68020 CPU's or with MC68000 CPU's with automatic stack growth the stack begins at location 0xe00000 and grows toward lower addresses. The stack is automatically extended as required.

On a Supermax computer equipped with MC68000 CPU's without automatic stack growth the stack begins at location 0xd00000 + stacksize and grows toward lower addresses. The stack is never extended.

The data segment is extended only as requested by the *brk*(2) system call.

For relocatable files the value of a word in the text or data portions that is not a reference to an undefined external symbol is exactly the value that will appear in memory when the file is executed. If a word in the text involves a reference to an undefined external symbol, there will be a relocation entry for the word, the storage class of the symbol-table entry for the symbol will be marked as an "external symbol", and the value and section number of the symbol-table entry will be undefined.

When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the word in the file.

File Header

The format of the **filehdr** header is

```
struct filehdr
{
        ushort  f_magic;    /* magic number */
        ushort  f_nscns;    /* number of sections */
        long    f_timdat;   /* time and date stamp */
        long    f_symptr;   /* file ptr to symtab */
        long    f_nsyms;    /* # symtab entries */
        ushort  f_opthdr;   /* sizeof(opt hdr) */
        ushort  f_flags;    /* flags */
};
```

UNIX System Header

The format of the UNIX system header is

```
typedef struct aouthdr
{
        short  magic;       /* magic number */
        short  vstamp;      /* version stamp */
        long   tsize;       /* text size in bytes, padded */
        long   dsize;       /* initialized data (.data) */
        long   bsize;       /* uninitialized data (.bss) */
        long   entry;       /* entry point */
        long   text_start;  /* base of text used for this file */
        long   data_start;  /* base of data used for this file */
} AOUTHDR;
```

Section Header

The format of the section header is

```
struct scnhdr
{
        char    s_name[SYMNMLEN];/* section name */
        long    s_paddr;     /* physical address */
        long    s_vaddr;     /* virtual address */
        long    s_size;      /* section size */
        long    s_scnptr;    /* file ptr to raw data */
        long    s_relptr;    /* file ptr to relocation */
        long    s_lnnoptr;   /* file ptr to line numbers */
        ushort  s_nreloc;    /* # reloc entries */
        ushort  s_nlnno;     /* # line number entries */
        long    s_flags;     /* flags */
};
```

Relocation

Object files have one relocation entry for each relocatable reference in the text or data. If relocation information is present, it will be in the following format:

```
struct reloc
{
        long    r_vaddr;     /* (virtual) address of reference */
        long    r_symndx;    /* index into symbol table */
        ushort  r_type;      /* relocation type */
};
```

The start of the relocation information is *s_relptr* from the section header. If there is no relocation information, *s_relptr* is 0.

Symbol Table

The format of each symbol in the symbol table is

```
#define  SYMNMLEN  8
#define  FILNMLEN  14
#define  DIMNUM    4

struct syment
{
    union  /* all ways to get a symbol name */
    {
        char     _n_name[SYMNMLEN]; /* name of symbol */
        struct
        {
            long  _n_zeroes;      /* == 0L if in string table */
            long  _n_offset;      /* location in string table */
        } _n_n;
        char     *_n_nptr[2]; /* allows overlaying */
    } _n;
    long        n_value;       /* value of symbol */
    short       n_scnum;       /* section number */
    ushort      n_type;        /* type and derived type */
    char        n_sclass;      /* storage class */
    char        n_numaux;      /* number of aux entries */
};

#define  n_name             _n._n_name
#define  n_zeroes           _n._n_n._n_zeroes
#define  n_offset           _n._n_n._n_offset
#define  n_nptr             _n._n_nptr[1]
```

Some symbols require more information than a single entry; they are followed by *auxiliary entries* that are the same size as a symbol entry. The format follows.

```
union auxent {
        struct {
                long     x_tagndx;
                union {
                        struct {
                                  unsigned short  x_lnno;
                                  unsigned short  x_size;
                        } x_lnsz;
                        long      x_fsize;
                } x_misc;
                union {
                        struct {
                                  long      x_lnnoptr;
                                  long      x_endndx;
                        }  x_fcn;
                        struct {
                                  unsigned short  x_dimen[DIMNUM];
                        } x_ary;
                } x_fcnary;
                unsigned short  x_tvndx;
        } x_sym;

        struct {
                char     x_fname[FILNMLEN];
        } x_file;

        struct {
                long         x_scnlen;
                unsigned short  x_nreloc;
                unsigned short  x_nlinno;
        } x_scn;

        struct {
                long             x_tvfill;
                unsigned short  x_tvlen;
                unsigned short  x_tvran[2];
        } x_tv;
};
```

Indexes of symbol table entries begin at *zero*. The start of the symbol table is *f_symptr* (from the file header) bytes from the beginning of the file. If the symbol table is stripped, *f_symptr* is 0. The string table (if one exists) begins at *f_symptr* + (*f_nsyms* ∗ SYMESZ) bytes from the beginning of the file.

SEE ALSO

as(1), cc(1), ld(1), brk(2), filehdr(4), ldfcn(4), linenum(4), reloc(4), scnhdr(4), syms(4).

This page is intentionally left blank

NAME

acct – per-process accounting file format

SYNOPSIS

#include <sys/acct.h>

DESCRIPTION

Files produced as a result of calling *acct*(2) have records in the form defined by <*sys/acct.h*>, whose contents are:

```
typedef  ushort comp_t;   /* "floating point" */
                          /* 13-bit fraction, 3-bit exponent  */


struct acct
{
        char     ac_flag;       /* Accounting flag */
        char     ac_stat;       /* Exit status */
        ushort   ac_uid;        /* Accounting user ID */
        ushort   ac_gid;        /* Accounting group ID */
        dev_t    ac_tty;        /* control typewriter */
        time_t   ac_btime;      /* Beginning time */
        comp_t   ac_utime;      /* acctng user time in clock ticks */
        comp_t   ac_stime;      /* acctng system time in clock ticks */
        comp_t   ac_etime;      /* acctng elapsed time in clock ticks */
        comp_t   ac_mem;        /* memory usage in clicks */
        comp_t   ac_io;         /* chars trnsfrd by read/write */
        comp_t   ac_rw;         /* number of block reads/writes */
        char     ac_comm[8];    /* command name */
};

extern   struct   acctacctbuf;
extern   struct   inode *acctp;  /* inode of accounting file */

#define   AFORK   01/* has executed fork, but no exec */
#define   ASU     02/* used super-user privileges */
#define   ACCTF   0300/* record type: 00 = acct */
```

In *ac_flag*, the AFORK flag is turned on by each *fork*(2) and turned off by an *exec*(2). The *ac_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds to *ac_mem* the current process size, computed as follows:

(data size) + (text size) / (number of in-core processes using text)

The value of *ac_mem* / *(ac_stime* + *ac_utime)* can be viewed as an approximation to the mean process size.

The structure acct, which resides with the source files of the accounting commands, represents the total accounting format used by the various accounting commands:

```
/*
 *  total accounting (for acct period), also for day
 */

struct          tacct {
     uid_t          ta_uid;        /* userid */
     char           ta_name[8];    /* login name */
     float          ta_cpu[2];     /* cum. cpu time, p/np (mins) */
     float          ta_kcore[2];   /* cum kcore-minutes, p/np */
     float          ta_con[2];     /* cum. connect time, p/np, mins */
     float          ta_du;         /* cum. disk usage */
     long           ta_pc;         /* count of processes */
     unsigned short ta_sc;         /* count of login sessions */
     unsigned short ta_dc;         /* count of disk samples */
     unsigned short ta_fee;        /* fee for special services */
};
```

SEE ALSO

acct(1M), acctcom(1M), acct(2), exec(2), fork(2).

BUGS

The *ac_mem* value for a short-lived command gives little information about the actual size of the command, because *ac_mem* may be incremented while a different command (e.g., the shell) is being executed by the process.

NAME

alphabet – alternative collation sequence files

DESCRIPTION

Files in **/usr/lib/alphabet** define alternative collation sequences used by the *straorder(3C)* function. The environment variable ALPHABET is the name of the file. Using an alternative collation sequence file makes it possible to sort files with strings in the ISO/DIS 8859/1 character set, in which the ordinal value of the character cannot be used as sorting criteria.

If the environment variable ALPHABET is not set, the ordinal value of the characters is used as sorting criteria as done in *strcmp*.

The format of the **/usr/lib/alphabet/$ALPHABET** in extended BNF syntax is as follows:

```
<alphabet> ::= /
               [ <ignore section> ]
               /
               [ <doubles section> ]
               /
               [ <collation section> ]
               /
               [ <comments section> ]
<ignore section> ::= <symbol> ; { <symbol> }
<doubles section> ::= <dbl def> ; { <dbl def> }
<dbl def> ::= <symbol> <symbol>
<collating section> ::= <plevel list> ; { <plevel list> }
<plevel list> ::= <slevel list> { <space> <slevel list> }
<slevel list> ::= <space list> | <equal list>
<space list> ::= <l2symbol> { <space> <l2symbol> }
<equal list> ::= <l2symbol> { = <l2symbol> }
```

```
<12symbol> ::=    <symbol> | <symbol> <symbol>

<symbol> ::=      <abs value> | <character>

<abs value> ::=   <decimal integer> |
                  0d <decimal integer> |
                  0x <hexidecimal integer> |
                  0o <octal integer> |
                  0b <binary integer>

<space> ::=       -- space character --
```

Ignore section

Characters listed in this section are ignored when two strings are compared. Normally characters 0x00-0x1f and 0x80-0x9f should be ignored.

Doubles section

This section lists characters that in a comparison should be processed as two adjacent characters. An example from the german alphabet would be:

```
ß=ss
```

meaning that whenever a character string contains the character sorted as the character string where 'ß' was replaced by 'ss'.

Collation sequence

This section defines the collation sequence and is a list of primary level definitions. For example:

```
/
/
/
a ;
b ;
c ;
/
```

means that the character 'a' precedes character 'b', and 'b' precedes 'c' ('precedes' is an associative relation) in the collation sequence.

Within each primary level a secondary level order may be defined. Single characters and two character strings may be ordered or given the same secondary ordinal value:

```
/
/
/
O=o Ó=ó Ō=ō ;
/
```

means that O, o, Ó, ó, Ō and ō alle have the same primary ordinal value, that O and o have the same secondary ordinal, but precedes Ó and ó, and so on.

The secondary ordinals are used only if two strings are found to be identical using the primary values.

Strings with identical primary and secondary values appear in arbitrary order on the sorted output.

The two character strings in the second level lists open for combinations like:

```
/
/
/
Å AA ;
å aa ;
/
```

used when sorting the Danish alphabet.

FILES

/usr/lib/alphabet – directory that contains the collating sequence files

SEE ALSO

sort(1), straorder(3C).

CAVEATS

The collation sequence has to be defined completely. Every character in the alphabet must either be ignored or placed in the collation sequence.

NAME

ar – common archive file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

The archive command *ar*(1) is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor *ld*(1).

Each archive begins with the archive magic string.

```
#define  ARMAG   "!<arch>\n"/* magic string */
#define  SARMAG  8          /* length of magic string */
```

Each archive which contains common object files [see *a.out*(4)] includes an archive symbol table. This symbol table is used by the link editor *ld*(1) to determine which archive members must be loaded during the link edit process. The archive symbol table (if it exists) is always the first file in the archive (but is never listed) and is automatically created and/or updated by *ar*.

Following the archive magic string are the archive file members. Each file member is preceded by a file member header which is of the following format:

```
#define  ARFMAG   "`\n"      /* header trailer string */

struct  ar_hdr          /* file member header */
{
    char    ar_name[16]; /* '/' terminated file member name */
    char    ar_date[12]; /* file member date */
    char    ar_uid[6];   /* file member user identification */
    char    ar_gid[6];   /* file member group identification */
    char    ar_mode[8];  /* file member mode (octal) */
    char    ar_size[10]; /* file member size */
    char    ar_fmag[2];  /* header trailer string */
};
```

All information in the file member headers is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for *ar_mode* which is in octal). Thus, if the archive contains printable files, the archive itself is printable.

The *ar_name* field is blank-padded and slash (/) terminated. The *ar_date* field is the modification date of the file at the time of its insertion into the archive. Common format archives can be moved from system to system as long as the portable archive command *ar*(1) is used. Conversion tools such as *convert*(1) exist to aid in the transportation of non-common format archives to this format.

Each archive file member begins on an even byte boundary; a newline is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

If the archive symbol table exists, the first file in the archive has a zero length name (i.e., **ar_name[0] = = '/'**). The contents of this file are as follows:

- The number of symbols. Length: 4 bytes.
- The array of offsets into the archive file. Length: 4 bytes * "the number of symbols".
- The name string table. Length: *ar_size* − (4 bytes * ("the number of symbols" + 1)).

The number of symbols and the array of offsets are managed with *sgetl* and *sputl*. The string table contains exactly as many null terminated strings as there are elements in the offsets array. Each offset from the array is associated with the corresponding name from the string table (in order). The names in the string table are all the defined global symbols found in the common object files in the archive. Each offset is the location of the archive header for the associated symbol.

SEE ALSO

ar(1), ld(1), strip(1), sputl(3X), a.out(4).

WARNINGS

strip(1) will remove all archive symbol entries from the header. The archive symbol entries must be restored via the **ts** option of the *ar*(1) command before the archive can be used with the link editor *ld*(1).

This page is intentionally left blank

NAME

cftime – language specific strings

DESCRIPTION

The programmer can create one printable file per language. These files must be kept in a special directory /lib/cftime. If this directory does not exist, the programmer should create it. The contents of these files are:

- abbreviated month names (in order)
- month names (in order)
- abbreviated weekday names (in order)
- weekday names (in order)
- default strings that specify formats for local time (%**x**) and local date (%**X**).
- default format for cftime, if the argument for cftime is zero or null.
- AM (ante meridian) string
- PM (post meridian) string

Each string is on a line by itself. All white space is significant. The order of the strings in the above list is the same order in which the strings appear in the file shown below.

EXAMPLE

/lib/cftime/usa_english

```
Jan
Feb
...
January
February
...
Sun
Mon
...
```

```
Sunday
Monday
 ...
%H:%M:%S
%m/%d/%y
%a %b %d %T %Z %Y
AM
PM
```

FILES

/lib/cftime – directory that contains the language specific printable files (create it if it does not exist)

SEE ALSO

ctime(3C).

NAME

checklist – list of file systems processed by fsck and ncheck

DESCRIPTION

checklist resides in directory **/etc** and contains a list of, at most, 15 *special file* names. Each *special file* name is contained on a separate line and corresponds to a file system. Each file system will then be automatically processed by the *fsck*(1M) command.

FILES

/etc/checklist

SEE ALSO

fsck(1M), ncheck(1M).

This page is intentionally left blank

NAME

core – format of core image file

SYNOPSIS

#include <sys/core.h>

DESCRIPTION

The Supermax Operating System writes out a core image of a terminated process when any of various errors occur. See *signal*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called **core** and is written in the process's working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The core file consists of four sections.

The first section is a copy of the local process control block, defined in **<sys/pl.h>**.

The second section is a copy of the text descriptor block, defined in **<sys/td.h>**.

The third section is an array of 12 memory allocation descriptors. They are either partition descriptors (described in **<sys/pd.h>**) or shared memory identifier descriptors (described in **<sys/SHM.h>**). The 12 descriptors describe memory segments 2 through 14, inclusive.

The fourth section is a copy of the process's data and stack segments.

SEE ALSO

sdb(1), signal(2).

This page is intentionally left blank

NAME

cpio − format of cpio archive

DESCRIPTION

The *header* structure, when the **−c** option of *cpio*(1) is not used, is:

```
struct {
        short   h_magic,
                h_dev;
        ushort  h_ino,
                h_mode,
                h_uid,
                h_gid;
        short   h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the **−c** option is used, the *header* information is described by:

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%11lo%s",
&Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
&Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
&Longtime, &Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file are recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

SEE ALSO

cpio(1), find(1), stat(2).

NAME

d_passwd – dial-up password file

DESCRIPTION

d_passwd contains for each program to use as shell the following information:

name of program to use as shell

encrypted password

This is an ASCII file. Each field within each program's entry is separated from the next by colon, the encrypted password further needs to be followed by a colon. Each program's entry is separated from the next by a new-line. If the password field is null, no password is required. Generation of encrypted password is done by the normal procedure (see *passwd*(1)). After generating the encrypted password in **/etc/passwd** is copied using a text editor.

This file resides in directory **/etc**. Because of the encrypted passwords it can have a general read permission.

FILES

/etc/d_passwd

SEE ALSO

login(1), passwd(1), dialups(4).

This page is intentionally left blank

NAME

dialups – dial-up connections

DESCRIPTION

dialups contains for each connections the following information:

name of connection

This is an ASCII file using tabs or spaces as field separator. Each connection's entry is separated from the next by a newline. The connection name is the full path name of the first pointer placed in the directory **/dev** to the dial-up connection.

This file resides in directory **/etc**.

FILES

/etc/dialups

SEE ALSO

login(1), d_passwd(4).

This page is intentionally left blank

NAME

dir – format of directories

SYNOPSIS

#include <sys/dir.h>

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry [see *fs*(4)]. The structure of a directory entry as given in the include file is:

```
#ifndef DIRSIZ
#define DIRSIZ      14
#endif
struct  direct
{
        ushort      d_ino;
        char   d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for **.** and **..**. The first is an entry for the directory itself. The second is for the parent directory. The meaning of **..** is modified for the root directory of the master file system; there is no parent, so **..** has the same meaning as **.**.

SEE ALSO

fs(4).

This page is intentionally left blank

NAME

dirent – file system independent directory entry

SYNOPSIS

#include <dirent.h>
#include <types.h>

DESCRIPTION

Different file system types may have different directory entries. The *dirent* structure defines a file system independent directory entry, which contains information common to directory entries in different file system types. A set of these structures is returned by the *getdents*(2) system call.

The *dirent* structure is defined below.

```
struct          dirent {
                long              d_ino;
                off_t             d_off;
                unsigned short    d_reclen;
                char              d_name[1];
};
```

The *d_ino* is a number which is unique for each file in the file system. The field *d_off* is the offset of that directory entry in the actual file system directory. The field *d_name* is the beginning of the character array giving the name of the directory entry. This name is null terminated and may have at most MAXNAMLEN characters. This results in file system independent directory entries being variable length entities. The value of *d_reclen* is the record length of this entry. This length is defined to be the number of bytes between the current entry and the next one, so that it will always result in the next entry being on a long boundary.

FILES

/usr/include/dirent.h

SEE ALSO

getdents(2).

This page is intentionally left blank

NAME

filehdr – file header for common object files

SYNOPSIS

#include <filehdr.h>

DESCRIPTION

Every common object file begins with a 20-byte header. The following C **struct** declaration is used:

```
struct filehdr
{
        unsigned short  f_magic ;   /* magic number */
        unsigned short  f_nscns ;   /* number of sections */
        long            f_timdat ;  /* time & date stamp */
        long            f_symptr ;  /* file ptr to symtab */
        long            f_nsyms ;   /* # symtab entries */
        unsigned short  f_opthdr ;  /* sizeof(opt hdr) */
        unsigned short  f_flags ;   /* flags */
} ;
```

f_symptr is the byte offset into the file at which the symbol table can be found. Its value can be used as the offset in *fseek*(3S) to position an I/O stream to the symbol table. The UNIX system optional header is 28 bytes long. The valid magic numbers are given below:

```
#define MC68MAGIC      0520   /* MC68000/MC68020 */
#define MC68020MAGIC   0526   /* MC68020 only */
```

The value in *f_timdat* is obtained from the *time*(2) system call. Flag bits currently defined are:

```
#define F_RELFLG   0000001  /* relocation entries stripped */
#define F_EXEC     0000002  /* file is executable */
#define F_LNNO     0000004  /* line numbers stripped */
#define F_LSYMS    0000010  /* local symbols stripped */
#define F_MINMAL   0000020  /* minimal object file */
#define F_UPDATE   0000040  /* update file, ogen produced */
#define F_SWABD    0000100  /* file is "pre-swabbed" */
```

```
#define F_AR16WR   0000200 /* 16-bit DEC host */
#define F_AR32WR   0000400 /* 32-bit DEC host */
#define F_AR32W    0001000 /* non-DEC host */
#define F_PATCH    0002000 /* "patch" list in opt hdr */
#define F_BM32ID   0160000 /* WE32000 family ID field */
#define F_BM32B    0020000 /* file contains WE 32100 code */
#define F_BM32MAU  0040000 /* file reqs MAU to execute */
#define F_BM32RST  0010000 /* this object file contains restore
                               work around [3B5/3B2 only] */
```

SEE ALSO

time(2), fseek(3S), a.out(4).

NAME

fs: file system – format of system volume

SYNOPSIS

#include <sys/types.h>
#include <sys/param.h>
#include <sys/fs/s5filsys.h>

DESCRIPTION

Every file system storage volume has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte long sectors. Sector 0 is unused and is available to contain a bootstrap program or other information.

Sector 1 is the *super-block*. The format of a super-block is:

```
struct    filsys
{
 ushort   s_isize;            /* size in blocks of i-list */
 daddr_t  s_fsize;            /* size in blocks of entire volume */
 short    s_nfree;            /* number of addresses in s_free */
 daddr_t  s_free[NICFREE];    /* free block list */
 short    s_ninode;           /* number of i-nodes in s_inode */
 ushort   s_inode[NICINOD];   /* free i-node list */
 char     s_flock;            /* lock during free list manipulation */
 char     s_ilock;            /* lock during i-list manipulation */
 char     s_fmod;             /* super block modified flag */
 char     s_ronly;            /* mounted read-only flag */
 time_t   s_time;             /* last super block update */
 short    s_dinfo[4];         /* device information */
 daddr_t  s_tfree;            /* total free blocks */
 ushort   s_tinode;           /* total free i-nodes */
 char     s_fname[6];         /* file system name */
 char     s_fpack[6];         /* file system pack name */
 long     s_fill[12];         /* ADJUST to make sizeof filsys
                                 be 512 */
 long     s_state;            /* file system state */
```

```
    long    s_magic;           /* magic number to denote new
                                  file system */
    long    s_type;            /* type of new file system */
};

#define FsMAGIC   0xfd187e20 /* s_magic number */

#define Fs1b      1          /* 512-byte block */
#define Fs2b      2          /* 1024-byte block */
#define Fs4b      3          /* 2048-byte block */

#define FsOKAY    0x7c269d38 /* s_state: clean */
#define FsACTIVE  0x5e72d81a /* s_state: active */
#define FsBAD     0xcb096f43 /* s_state: bad root */
#define FsBADBLK  0xbadbc14b /* s_state: bad block corrupted it */
```

s_type indicates the file system type. Currently, two types of file systems are normally used in Supermax systems: The original 512-byte logical block and the improved 2048-byte logical block. *s_magic* is used to distinguish the original 512-byte oriented file systems from the newer file systems. If this field is not equal to the magic number, *FsMAGIC*, the type is assumed to be *Fs1b*, otherwise the *s_type* field is used. In the following description, a block is then determined by the type. For the original 512-byte oriented file system, a block is 512-bytes. For the 2048-byte oriented file system, a block is 2048-bytes or four sectors. The operating system takes care of all conversions from logical block numbers to physical sector numbers.

s_state indicates the state of the file system. A cleanly unmounted, not damaged file system is indicated by the FsOKAY state. After a file system has been mounted for update, the state changes to FsACTIVE. A special case is used for the root file system. If the root file system appears damaged at boot time, it is mounted but marked FsBAD. Lastly, after a file system has been unmounted, the state reverts to FsOKAY.

s_isize is the address of the first data block after the i-list; the i-list starts just after the super-block, namely in block 2; thus the i-list is *s_isize* − 2 blocks long. *s_fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an "impossible" block number is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free*[1], . . ., *s_free*[*s_nfree* − 1], up to 49 numbers of free blocks. *s_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block is *s_free*[*s_nfree*]. If the new block number is 0, there are no blocks left, so give an error. If *s_nfree* became 0, read in the block named by the new block number, replace *s_nfree* by its first word, and copy the block numbers in the next 50 longs into the *s_free* array. To free a block, check if *s_nfree* is 50; if so, copy *s_nfree* and the *s_free* array into it, write it out, and set *s_nfree* to 0. In any event set *s_free*[*s_nfree*] to the freed block's number and increment *s_nfree*.

s_tfree is the total free blocks available in the file system.

s_ninode is the number of free i-numbers in the *s_inode* array. To allocate an i-node: if *s_ninode* is greater than 0, decrement it and return *s_inode*[*s_ninode*]. If it was 0, read the i-list and place the numbers of all free i-nodes (up to 100) into the *s_inode* array, then try again. To free an i-node, provided *s_ninode* is less than 100, place its number into *s_inode*[*s_ninode*] and increment *s_ninode*. If *s_ninode* is already 100, do not bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the i-node is really free or not is maintained in the i-node itself.

s_tinode is the total free i-nodes available in the file system.

s_flock and *s_ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *s_fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

s_ronly is a read-only flag to indicate write-protection.

s_time is the last time the super-block of the file system was changed, and is the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT). During a reboot, the *s_time* of the super-block for the root file system is used to set the system's idea of the time.

s_fname is the name of the file system and *s_fpack* is the name of the pack.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 64 bytes long. I-node 1 is reserved for future use. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. For the format of an i-node and its flags, see *inode*(4).

SEE ALSO

fsck(1M), fsdb(1M), mkfs(1M), mount(2), inode(4).

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the UNIX system with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX system commands.

A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

45

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets **<:** and **:>**. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a – followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a – followed by the name of a "canned" tab specification.

Standard tabs are specified by **t–8**, or equivalently, **t1,9,17,25,**etc. The canned tabs which are recognized are defined by the *tabs*(1) command.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer.

Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin*
The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d
The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e
The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t−8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

```
* <:t5,10,15 s72:> *
```

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

SEE ALSO

ed(1), newform(1), tabs(1).

NAME

fstab – file-system-table

DESCRIPTION

The **/etc/fstab** file contains information about file systems for use by **mount (1M)** and **mountall(1M).** Each entry in **/etc/fstab** has the following format:

column 1	block special file name of file system or advertised remote resource
column 2	mount-point directory
column 3	"–r" if to be mounted read-only; "–d[r]" if remote
column 4	(optional) file system type string
column 5+	ignored

White-space separates columns. Lines beginning with "# " are comments. Empty lines are ignored.

A file-system-table might read:

```
/dev/dsk/c1d0s2 /usr  S51K
/dev/dsk/c1d1s2 /usr/src  –r
adv_resource /mnt  –d
```

FILES

/etc/fstab

SEE ALSO

mount(1M), mountall(1M), rmountall(1M).

This page is intentionally left blank

NAME

gettydefs – speed and terminal settings used by getty

DESCRIPTION

The **/etc/gettydefs** file contains information used by *getty*(1M) to set up the speed and terminal settings for a line. It supplies information on what the *login*(1) prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a <*break*> character.

Each entry in **/etc/gettydefs** has the following format:

label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. The various fields can contain quoted characters of the form **\b**, **\n**, **\c**, etc., as well as \nnn, where *nnn* is the octal value of the desired character. The various fields are:

label	This is the string against which *getty*(1M) tries to match its second argument. It is often the speed, such as **1200**, at which the terminal is supposed to run, but it need not be (see below).
initial-flags	These flags are the initial *ioctl*(2) settings to which the terminal is to be set if a terminal type is not specified to *getty*(1M). The flags that *getty*(1M) understands are the same as the ones listed in **/usr/include/sys/termio.h** [see *termio*(7)]. Normally only the speed flag is required in the *initial-flags*. *getty*(1M) automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty*(1M) executes *login*(1).
final-flags	These flags take the same values as the *initial-flags* and are set just before *getty*(1M) executes *login*(1). The speed flag is again required. The composite flag **SANE** takes care

of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. **SANE8** can be used instead of **SANE** to specify 8-bit operation. **DDE_CTL** can be used to specify the commonly used DDE control characters. Other commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL**, so that the line is hung up on the final close.

login-prompt This entire field is printed as the *login-prompt*. Unlike the above fields where white space is ignored (a space, tab or new-line), they are included in the *login-prompt* field.

next-label If this entry does not specify the desired speed, indicated by the user typing a <*break*> character, then *getty*(1M) will search for the entry with *next-label* as its *label* field and set up the terminal for those settings. Usually, a series of speeds are linked together in this fashion, into a closed set; for instance, **2400** linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

If *getty*(1M) is called without a second argument, then the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. It is also used if *getty*(1M) can not find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into *getty*(1M) which will bring up a terminal at **300** baud.

It is strongly recommended that after making or modifying **/etc/gettydefs**, it be run through *getty*(1M) with the check option to be sure there are no errors.

FILES

/etc/gettydefs

SEE ALSO

getty(1M), login(1), stty(1), ioctl(2), termio(7).

BUGS

8-bit with parity mode is not supported.

This page is intentionally left blank

NAME

group – group file

DESCRIPTION

group contains for each group the following information:

group name
encrypted password
numerical group ID
comma-separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

newgrp(1M), passwd(1), passwd(4).

This page is intentionally left blank

NAME

inittab – script for the init process

DESCRIPTION

The *inittab* file supplies the script to *init*'s role as a general process dispatcher. The process that constitutes the majority of *init*'s process dispatching activities is the line process **/etc/getty** that initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

The *inittab* file is composed of entries that are position dependent and have the following format:

```
id:rstate:action:process
```

Each entry is delimited by a newline, however, a backslash (\\) preceding a newline indicates a continuation of the entry. Up to 512 characters per entry are permitted. Comments may be inserted in the *process* field using the *sh*(1) convention for comments. Comments for lines that spawn *getty*s are displayed by the *who*(1) command. It is expected that they will contain some information about the line such as the location. There are no limits (other than maximum entry size) imposed on the number of entries within the *inittab* file. The entry fields are:

id This is one or two characters used to uniquely identify an entry.

rstate This defines the *run-level* in which this entry is to be processed. *run-levels* effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a *run-level* or *run-levels* in which it is allowed to exist. The *run-levels* are represented by a number ranging from **0** through **6**. As an example, if the system is in *run-level* **1**, only those entries having a **1** in the *rstate* field will be processed. When *init* is requested to change *run-levels,* all processes which do not have an entry in the *rstate* field for the target *run-level* will be sent the warning signal (**SIGTERM**) and allowed a 20-second grace period before being forcibly terminated by a kill signal (**SIGKILL**). The *rstate* field can define multiple *run-levels* for a process by selecting more than one *run-level* in any combination from **0**−**6**. If no *run-level* is specified, then the process is assumed to be valid at all *run-levels* **0**−**6**. There are three other values, **a**, **b** and **c**, which can appear in the *rstate* field, even though they are not true *run-levels*. Entries which have these characters in the *rstate* field are processed only when the *telinit* [see *init*(1M)] process requests them to be run (regardless of the current *run-level* of the system). They differ from *run-levels* in that *init* can never enter *run-level* **a**, **b** or **c**. Also, a request for the execution of any of these processes does not change the current *run-level*. Furthermore, a process started by an **a**, **b** or **c** command is not killed when *init* changes levels. They are only killed if their line in **/etc/inittab** is marked **off** in the *action* field, their line is deleted entirely from **/etc/inittab**, or *init* goes into the *SINGLE USER* state.

action Key words in this field tell *init* how to treat the process specified in the *process* field. The actions recognized by *init* are as follows:

respawn If the process does not exist then start the process, do not wait for its termination (continue scanning the *inittab* file), and when it dies restart the process. If the process currently exists then do nothing and continue scanning the *inittab* file.

wait Upon *init*'s entering the *run-level* that matches the entry's *rstate*, start the process and wait for its termination. All subsequent reads of the *inittab* file while *init* is in the same *run-level* will cause *init* to ignore this entry.

once	Upon *init*'s entering a *run-level* that matches the entry's *rstate*, start the process, do not wait for its termination. When it dies, do not restart the process. If upon entering a new *run-level*, where the process is still running from a previous *run-level* change, the program will not be restarted.
boot	The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *init* is to start the process, not wait for its termination; and when it dies, not restart the process. In order for this instruction to be meaningful, the *rstate* should be the default or it must match *init*'s *run-level* at boot time. This action is useful for an initialization function following a hardware reboot of the system.
bootwait	The entry is to be processed the first time *init* goes from single-user to multi-user state after the system is booted. (If **initdefault** is set to **2**, the process will run right after the boot.) *Init* starts the process, waits for its termination and, when it dies, does not restart the process.
powerfail	Execute the process associated with this entry only when *init* receives a power fail signal [**SIGPWR** see *signal*(2)].
powerwait	Execute the process associated with this entry only when *init* receives a power fail signal (**SIGPWR**) and wait

until it terminates before continuing any processing of *inittab*.

off	If the process associated with this entry is currently running, send the warning signal (**SIGTERM**) and wait 20 seconds before forcibly terminating the process via the kill signal (**SIGKILL**). If the process is nonexistent, ignore the entry.
ondemand	This instruction is really a synonym for the **respawn** action. It is functionally identical to **respawn** but is given a different keyword in order to divorce its association with *run-levels*. This is used only with the **a**, **b** or **c** values described in the *rstate* field.
initdefault	An entry with this *action* is only scanned when *init* initially invoked. *Init* uses this entry, if it exists, to determine which *run-level* to enter initially. It does this by taking the highest *run-level* specified in the **rstate** field and using that as its initial state. If the *rstate* field is empty, this is interpreted as **0123456** and so *init* will enter *run-level* **6**. Additionally, if *init* does not find an **initdefault** entry in **/etc/inittab**, then it will request an initial *run-level* from the user at reboot time.
sysinit	Entries of this type are executed before *init* tries to access the console (i.e., before the **Console Login:** prompt). It is expected that this entry will be only used to initialize devices on which *init* might try to ask

the *run-level* question. These entries are executed and waited for before continuing.

process This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh −c 'exec** *command'*. For this reason, any legal *sh* syntax can appear in the *process* field. Comments can be inserted with the **;** **#***comment* syntax.

FILES

/etc/inittab

SEE ALSO

getty(1M), init(1M), sh(1), who(1), exec(2), open(2), signal(2).

This page is intentionally left blank

NAME

inode – format of an i-node

SYNOPSIS

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

DESCRIPTION

An i-node for a plain file or directory in a file system has the following structure defined by **<sys/ino.h>**.

```
/* Inode structure as it appears on a disk block. */
struct  dinode
{
ushort  di_mode;        /* mode and type of file */
short   di_nlink;       /* number of links to file */
ushort  di_uid;         /* owner's user id */
ushort  di_gid;         /* owner's group id */
off_t   di_size;        /* number of bytes in file */
char    di_addr[40];    /* disk block addresses */
time_t  di_atime;       /* time last accessed */
time_t  di_mtime;       /* time last modified */
time_t  di_ctime;       /* time of last file status change */
};
/*
 * the 40 address bytes:
 *      39 used; 13 addresses
 *      of 3 bytes each.
 */
```

For the meaning of the defined types *off_t* and *time_t* see *types*(5).

SEE ALSO

stat(2), fs(4), types(5).

This page is intentionally left blank

NAME

issue – issue identification file

DESCRIPTION

The file **/etc/issue** contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file which is read by program *getty* and then written to any terminal spawned or respawned from the *lines* file.

FILES

/etc/issue

SEE ALSO

login(1).

This page is intentionally left blank

NAME

ldfcn – common object file access routines

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

DESCRIPTION

The common object file access routines are a collection of functions for reading common object files and archives containing common object files. Although the calling program must know the detailed structure of the parts of the object file that it processes, the routines effectively insulate the calling program from knowledge of the overall structure of the object file.

The interface between the calling program and the object file access routines is based on the defined type **LDFILE**, defined as **struct ldfile**, declared in the header file **ldfcn.h**. The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function *ldopen*(3X) allocates and initializes the **LDFILE** structure and returns a pointer to the structure to the calling program. The fields of the **LDFILE** structure may be accessed individually through macros defined in **ldfcn.h** and contain the following information:

LDFILE	* ldptr;
TYPE(ldptr)	The file magic number used to distinguish between archive members and simple object files.
IOPTR(ldptr)	The file pointer returned by *fopen* and used by the standard input/output functions.
OFFSET(ldptr)	The file address of the beginning of the object file; the offset is non-zero if the object file is a member of an archive file.

HEADER(ldptr) The file header structure of the object file.

The object file access functions themselves may be divided into four categories:

(1) functions that open or close an object file

ldopen(3X) and *ldaopen*[see *ldopen*(3X)]
open a common object file

ldclose(3X) and *ldaclose*[see *ldclose*(3X)]
close a common object file

(2) functions that read header or symbol table information

ldahread(3X)
read the archive header of a member of an archive file

ldfhread(3X)
read the file header of a common object file

ldshread(3X) and *ldnshread*
[see *ldshread*(3X)]
read a section header of a common object file

ldtbread(3X)
read a symbol table entry of a common object file

ldgetname(3X)
retrieve a symbol name from a symbol table entry or from the string table

(3) functions that position an object file at (seek to) the start of the section, relocation, or line number information for a particular section.

ldohseek(3X)
seek to the optional file header of a common object file

ldsseek(3X) and *ldnsseek*[see *ldsseek*(3X)]
: seek to a section of a common object file

ldrseek(3X) and *ldnrseek*[see *ldrseek*(3X)]
: seek to the relocation information for a section of a common object file

ldlseek(3X) and *ldnlseek*[see *ldlseek*(3X)]
: seek to the line number information for a section of a common object file

ldtbseek(3X)
: seek to the symbol table of a common object file

(4) the function *ldtbindex*(3X) which returns the index of a particular common object file symbol table entry.

These functions are described in detail on their respective manual pages.

All the functions except *ldopen*(3X), *ldgetname*(3X), *ldtbindex*(3X) return either **SUCCESS** or **FAILURE**, both constants defined in **ldfcn.h**. *ldopen*(3X) and *ldaopen*[(see *ldopen*(3X)] both return pointers to an **LDFILE** structure.

Additional access to an object file is provided through a set of macros defined in **ldfcn.h**. These macros parallel the standard input/output file reading and manipulating functions, translating a reference of the **LDFILE** structure into a reference to its file descriptor field.

The following macros are provided:

```
GETC(ldptr)
FGETC(ldptr)
GETW(ldptr)
UNGETC(c, ldptr)
FGETS(s, n, ldptr)
FREAD((char * ) ptr, sizeof ( * ptr), nitems, ldptr)
FSEEK(ldptr, offset, ptrname)
FTELL(ldptr)
```

```
REWIND(ldptr)
FEOF(ldptr)
FERROR(ldptr)
FILENO(ldptr)
SETBUF(ldptr, buf)
STROFFSET(ldptr)
```

The STROFFSET macro calculates the address of the string table. See the manual entries for the corresponding standard input/output library functions for details on the use of the rest of the macros.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fseek(3S), ldahread(3X), ldclose(3X), ldgetname(3X), ldfhread(3X), ldlread(3X), ldlseek(3X), ldohseek(3X), ldopen(3X), ldrseek(3X), ldlseek(3X), ldshread(3X), ldtbindex(3X), ldtbread(3X), ldtbseek(3X), stdio(3S), intro(5).

WARNING

The macro **FSEEK** defined in the header file **ldfcn.h** translates into a call to the standard input/output function *fseek*(3S). **FSEEK** should not be used to seek from the end of an archive file since the end of an archive file may not be the same as the end of one of its object file members!

NAME

limits – file header for implementation-specific constants

SYNOPSIS

#include <limits.h>

DESCRIPTION

The header file <*limits.h*> is a list of magnitude limitations imposed by a specific implementation of the operating system. All values are specified in decimal.

```
#define ARG_MAX   5120   /* max length of arguments to exec */
#define CHAR_BIT  8      /* # of bits in a "char" */
#define CHAR_MAX  127    /* max integer value of a "char" */
#define CHAR_MIN  -128   /* min integer value of a "char" */
#define CHILD_MAX 1024   /* max # of processes per user id */
#define CLK_TCK   50     /* # of clock ticks per second */
#define DBL_DIG   15     /* digits of precision of a "double" */
#define DBL_MAX   1.79769313486231470e+308
                         /*max decimal value of a "double"*/
#define DBL_MIN   4.94065645841246544e-324
                         /*min decimal value of a "double"*/
#define FCHR_MAX  1048576
                         /*max size of a file in bytes */
#define FLT_DIG   7      /*digits of precision of a "float" */
#define FLT_MAX   3.40282346638528860e+38
                         /*max decimal value of a "float" */
#define FLT_MIN   1.40129846432481707e-45
                         /*min decimal value of a "float" */
#define HUGE_VAL  3.40282346638528860e+38
                         /*error value returned by Math lib*/
#define INT_MAX   2147483647
                         /*max decimal value of an "int" */
#define INT_MIN   -2147483648
                         /*min decimal value of an "int" */
#define LINK_MAX  1000   /*max # of links to a single file */
#define LONG_MAX  2147483647
                         /*max decimal value of a "long" */
```

```
#define LONG_MIN   -2147483648
                          /*min decimal value of a "long" */
#define NAME_MAX   14     /*max # of characters in a file name */
#define OPEN_MAX   32     /*max # files a process can have open */
#define PASS_MAX   8      /*max # of characters in a password */
#define PATH_MAX   512    /*max # of characters in a path name */
#define PID_MAX    32767  /*max value for a process ID */
#define PIPE_BUF   5120   /*max # bytes atomic in write to a pipe */
#define PIPE_MAX   5120   /*max # bytes written to a pipe in a write */
#define SHRT_MAX   32767  /*max decimal value of a "short" */
#define SHRT_MIN   -32768
                          /*min decimal value of a "short" */
#define STD_BLK    2048   /* # bytes in a physical I/O block */
#define SYS_NMLN   9      /* # of chars in uname-returned strings */
#define UID_MAX    60000  /*max value for a user or group ID */
#define USI_MAX    4294967295
                          /*max decimal value of an "unsigned" */
#define WORD_BIT   32     /* # of bits in a "word" or "int" */
```

70

NAME

linenum – line number entries in a common object file

SYNOPSIS

#include <linenum.h>

DESCRIPTION

The *cc* command generates an entry in the object file for each C source line on which a breakpoint is possible [when invoked with the **–g** option; see *cc*(1)]. Users can then reference line numbers when using the appropriate software test system [see *sdb*(1)]. The structure of these line number entries appears below.

```
struct  lineno
{
        union
        {
                long    l_symndx ;
                long    l_paddr ;
        }               l_addr ;
        unsigned short  l_lnno ;
} ;
```

Numbering starts with one for each function. The initial line number entry for a function has *l_lnno* equal to zero, and the symbol table index of the function's entry is in *l_symndx*. Otherwise, *l_lnno* is non-zero, and *l_paddr* is the physical address of the code for the referenced line. Thus the overall structure is the following:

```
l_addr                    l_lnno

function symtab index     0
physical address          line
physical address          line
...
function symtab index     0
physical address          line
physical address          line
...
```

SEE ALSO

cc(1), sdb(1), a.out(4).

NAME

mnttab – mounted file system table

SYNOPSIS

#include <mnttab.h>

DESCRIPTION

mnttab resides in directory **/etc** and contains a table of devices, mounted by the *mount*(1M) command, in the following structure as defined by **<mnttab.h>**:

```
struct    mnttab {
          char          mt_dev[32];
          char          mt_filsys[32];
          short         mt_ro_flg;
          time_t        mt_time;
};
```

Each entry is 70 bytes in length; the first 32 bytes are the null-padded name of the place where the *special file* is mounted; the next 32 bytes represent the null-padded root name of the mounted special file; the remaining 6 bytes contain the mounted *special file*'s read/write permissions and the date on which it was mounted.

SEE ALSO

mount(1M), setmnt(1M).

This page is intentionally left blank

NAME

passwd − password file

DESCRIPTION

passwd contains for each user the following information:

login name
encrypted password
numerical user ID
numerical group ID
GCOS job number, box number, optional GCOS user ID
initial working directory
program to use as shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the shell field is null, the shell itself is used.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user IDs to names.

The encrypted password consists of 13 characters chosen from a 64-character alphabet (**.**, **/**, **0**−**9**, **A**−**Z**, **a**−**z**), except when the password is null, in which case the encrypted password is also null. Password aging is effected for a particular user if his encrypted password in the password file is followed by a comma and a non-null string of characters from the above alphabet. (Such a string must be introduced in the first instance by the super-user.)

The first character of the age, *M* say, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after his password has expired will be forced to supply a new one. The next character, *m* say, denotes the minimum period in weeks which must expire before the password may be changed. The remaining characters define the week (counted from the beginning of 1970) when the password was last changed. (A null string is equivalent to zero.) *M* and *m* have numerical values in the range 0–63 that correspond to the 64-character alphabet shown above (i.e., **/** = 1 week; **z** = 63 weeks). If *m* = *M* = 0 (derived from the string **.** or **..**) the user will be forced to change his password the next time he logs in (and the "age" will disappear from his entry in the password file). If *m* > *M* (signified, e.g., by the string **./**) only the super-user will be able to change the password.

The recommended use of the GCOS is:

user name, options

The user name will be used by some utilities in the future. The options do not need to be specified. The options known and used by *login*(1) is as follows:

pri=<number>	(see *nice*(1))
ulimit=<number>	(see *ulimit*(1))
mcumask=<number>	(see *mcumask*(1))

The options **pri=** and **ulimit=** reads <*number*> using decimal notation, and **mcumask=** reads the <*number*> in octal notation.

FILES

/etc/passwd

SEE ALSO

login(1), passwd(1), a64l(3C), getpwent(3C), group(4).

NAME

profile – setting up an environment at login time

SYNOPSIS

/etc/profile
$HOME/.profile

DESCRIPTION

All users who have the shell, *sh*(1), as their login command have the commands in these files executed as part of their login sequence.

/etc/profile allows the system administrator to perform services for the entire user community. Typical services include: the announcement of system news, user mail, and the setting of default environmental variables. It is not unusual for */etc/profile* to execute special actions for the **root** login or the *su*(1) command. Computers running outside the Eastern time zone should have the line

```
        . /etc/TIMEZONE
```

included early in */etc/profile* (see timezone(4)).

The file *$HOME/.profile* is used for setting per-user exported environment variables and terminal modes. The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 027
# Tell me when new mail comes in
MAIL=/usr/mail/$LOGNAME
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
# Set terminal type
while :
do  echo "terminal: \c"
    read TERM
    if [ -f ${TERMINFO:-/usr/lib/terminfo}/?/$TERM ]
```

```
          then break
          elif [ −f /usr/lib/terminfo/?/$TERM ]
          then break
          else echo "invalid term $TERM" 1>&2
          fi
     done
     # Initialize the terminal and set tabs
     # The environmental variable TERM must have been
     #  exported before the "tput init" command is executed.
     tput init
     # Set the erase character to backspace
     stty erase '^H' echoe
```

FILES

/etc/TIMEZONE	timezone environment
$HOME/.profile	user – specific environment
/etc/profile	system – wide environment

SEE ALSO

env(1), login(1), mail(1), sh(1), stty(1), su(1M), terminfo(4), timezone(4), environ(5), term(5).

User's Guide.

Chapter 10 in the *Programmer's Guide.*

NOTES

Care must be taken in providing system-wide services in */etc/profile*. Personal *.profile* files are better for serving all but the most global needs.

NAME

proto – prototype job file for at

SYNOPSIS

/usr/lib/cron/.proto

/usr/lib/.proto.*queue*

DESCRIPTION

When a job is submitted to *at*(1) or *batch*(1), the job is constructed as a shell script. First, a prologue is constructed, consisting of:

- A header whether the job is an *at* job or a *batch* job (actually, *at* jobs submitted to all queues other than queue **a**, not just the batch queue **b**, are listed as *batch* jobs); the header will be:

  : at job

  for an *at* job, and

  : batch job

  for a *batch* job.

- A set of Bourne shell commands to make the environment (see *environ*(5)) for the *at* job the same as the current environment.

- A command to run the user's shell (as specified by the environment variable) with the rest of the job file as input.

at then reads a prototype file, and constructs the rest of the job file from it.

Text from the prototype file is copied to the job file, except for special variables that are replaced by other text:

$d	is replaced by the current working directory.
$l	is replaced by the current file size limit (see *ulimit*(2)).
$m	is replaced by the current umask (see *umask*(2)).

$t is replaced by the time at which the job should be run, expressed as seconds since January 1, 1970, 00:00 Greenwich Mean Time, preceded by a colon.

$< is replaced by text read by *at* from the standard input (that is, the commands provided to *at* to be run in the job).

If the job is submitted in queue *queue*, *at* uses the file **/usr/lib/cron/.proto.***queue* as the prototype file if it exists, otherwise it will use the file **/usr/lib/cron/.proto**.

EXAMPLE

The standard **.proto** file supplied is:

```
#ident"@(#)adm:proto    1.2"
cd $d
ulimit $1
umask $m
$<
```

which causes commands to change the current directory in the job to the current directory at the time *at* was run, to change the file size limit in the job to the file size limit at the time *at* was run, and to change the umask in the job to the umask at the time *at* was run, to be inserted before the commands in the job.

FILES

/usr/lib/cron/proto
/usr/lib/cron/.proto.*queue*

SEE ALSO

at(1), ulimit(2), umask(2), environ(5).

81

NAME

queuedefs – at/batch/cron queue description file

SYNOPSIS

/usr/lib/cron/queuedefs

DESCRIPTION

The *queuedefs* file describes the characteristics of the queues managed by *cron*(1). Each non-comment line in this file describes one queue. The format of the lines are as follows:

```
q.[njobj][nicen][nwaitw]
```

The fields in this line are:

q	The name of the queue. **a** is the default queue for jobs started by *at*(1); **b** is the default queue for jobs started by *batch*(1); **c** is the default queue for jobs run from a **crontab** file.
njob	The maximum number of jobs that can be run simultaneously in that queue; if more than *njob* jobs are ready to run, only the first *njob* jobs will run, and the others will be run, as jobs – that are currently running – terminate. The default value is 100.
nice	The *nice*(1) value to give all jobs in that queue that are not run with a user ID of super-user. The default value is 2.
nwait	The number of seconds to wait before rescheduling a job that was deferred because more than *njob* jobs were running in that job's queue, or because more than 25 jobs were running in all the queues. The default value is 60.

Lines beginning with **#** are comments, and are ignored.

EXAMPLE

```
a.4jln
b.2j2n90w
```

This file specifies that the **a** queue, for *at* jobs, can have up to 4 jobs running simultaneously; those jobs will be run with a *nice* value of 1. As no *nwait* value was given, if a job cannot be run because too many other jobs are running, *cron* will wait 60 seconds before trying again to run it. The **b** queue, for *batch* jobs, can have up to 2 jobs running simultaneously; those jobs will be run with a *nice* value of 2. If a job cannot be run because too many other jobs are running, *cron* will wait 90 seconds before trying again to run it. All other queues can have up to 100 jobs running simultaneously; they will be run with a *nice* value of 2, and if a job cannot be run because too many other jobs are running, *cron* will wait 60 seconds before trying again to run it.

FILES

/usr/lib/cron/queuedefs

SEE ALSO

cron(1).

NAME

reloc – relocation information for a common object file

SYNOPSIS

#include <reloc.h>

DESCRIPTION

Object files have one relocation entry for each relocatable reference in the text or data. If relocation information is present, it will be in the following format.

```
struct      reloc
{
    long    r_vaddr ;    /* (virtual) address of reference */
    long    r_symndx ;   /* index into symbol table */
    ushort  r_type ;     /* relocation type */
} ;

#define     R_ABS       0
#define     R_DIR32     06
#define     R_DIR32S    012
```

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

R_ABS	The reference is absolute and no relocation is necessary. The entry will be ignored.
R_DIR32	A direct 32-bit reference to the symbol's virtual address.
R_DIR32S	A direct 32-bit reference to the symbol's virtual address, with the 32-bit value stored in the reverse order in the object file.

More relocation types exist for other processors. Equivalent relocation types on different processors have equal values and meanings. New relocation types will be defined (with new values) as they are needed.

Relocation entries are generated automatically by the assembler and automatically used by the link editor. Link editor options exist for both preserving and removing the relocation entries from object files.

SEE ALSO

as(1), ld(1), a.out(4), syms(4).

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS (Source Code Control System) file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the **ASCII SOH** (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form **DDDDD** represent a five-digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

@hDDDDD

The value of the checksum is the sum of all characters, except those of the first line. The **@h** provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgrm> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
.
.
.
@c <comments> ...
.
.
.
@e
```

The first line (**@s**) contains the number of lines inserted/deleted/unchanged, respectively. The second line (**@d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@m** lines (optional) each contain one **MR** number associated with the delta; the **@c** lines contain comments associated with the delta.

The **@e** line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @**u** and @**U**. An empty list allows anyone to make a delta. Any line starting with a **!** prohibits the succeeding group or user from making deltas.

Flags

Keywords used internally. [See *admin*(1) for more information on their use.] Each flag line takes the form:

@**f** <flag> <optional text>

The following flags are defined:

@**f** t<type of program>
@**f** v<program name>
@**f** i<keyword string>
@**f** b
@**f** m<module name>
@**f** f<floor>
@**f** c<ceiling>
@**f** d<default-sid>
@**f** n
@**f** j
@**f** l<lock-releases>
@**f** q<user defined>
@**f** z<reserved for use in interfaces>

The **t** flag defines the replacement for the **%Y%** identification keyword. The **v** flag controls prompting for **MR** numbers in addition to comments; if the optional text is present it defines an **MR** number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **−b** keyletter may be used on the *get* command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the **%M%** identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing [*get*(1) with the **−e** keyletter]. The **q** flag defines the replacement for the **%Q%** identification keyword. The **z** flag is used in certain specialized interface programs. *Comments* Arbitrary text is surrounded by the bracketing lines @**t** and @**T**. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines do not begin with the control character, control lines do. There are three kinds of control lines: *insert*, *delete*, and *end*, represented by:

```
@I DDDDD
@D DDDDD
@E DDDDD
```

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).

NAME

scnhdr – section header for a common object file

SYNOPSIS

#include <scnhdr.h>

DESCRIPTION

Every common object file has a table of section headers to specify the layout of the data within the file. Each section within an object file has its own header. The C structure appears below.

```
struct  scnhdr
{
  char   s_name[SYMNMLEN]; /* section name */
  long   s_paddr;          /* physical address */
  long   s_vaddr;          /* virtual address */
  long   s_size;           /* section size */
  long   s_scnptr;         /* file ptr to raw data */
  long   s_relptr;         /* file ptr to relocation */
  long   s_lnnoptr;        /* file ptr to line numbers */
  unsigned short  s_nreloc;  /* # reloc entries */
  unsigned short  s_nlnno;   /* # line number entries */
  long   s_flags;          /* flags */
};
```

File pointers are byte offsets into the file; they can be used as the offset in a call to FSEEK [see *ldfcn*(4)]. If a section is initialized, the file contains the actual bytes. An uninitialized section is somewhat different. It has a size, symbols defined in it, and symbols that refer to it. But it can have no relocation entries, line numbers, or data. Consequently, an uninitialized section has no raw data in the object file, and the values for *s_scnptr*, *s_relptr*, *s_lnnoptr*, *s_nreloc*, and *s_nlnno* are zero.

SEE ALSO

ld(1), fseek(3S), a.out(4), ldfcn(4).

This page is intentionally left blank

NAME

scr_dump – format of curses screen image file.

SYNOPSIS

scr_dump(file)

DESCRIPTION

The *curses*(3X) function *scr_dump*() will copy the contents of the screen into a file. The format of the screen image is as described below.

The name of the tty is 20 characters long and the modification time (the *mtime* of the tty that this is an image of) is of the type *time_t*. All other numbers and characters are stored as *chtype* (see **<curses.h>**). No newlines are stored between fields.

```
<magic number:  octal 0433>
<name of tty>
<mod time of tty>
<columns>  <lines>
<line length>  <chars in line>    for each line on the screen
<line length>  <chars in line>
.
.
<labels?>  1, if soft screen labels are present
<cursor row>  <cursor column>
```

Only as many characters as are in a line will be listed. For example, if the *<line length>* is **0**, there will be no characters following *<line length>*. If *<labels?>* is TRUE, following it will be

```
<number of labels>
<label width>
<chars in label 1>
<chars in label 2>
.
.
```

SEE ALSO

curses(3X).

NAME

syms – common object file symbol table format

SYNOPSIS

#include <syms.h>

DESCRIPTION

Common object files contain information to support symbolic software testing [see *sdb*(1)]. Line number entries, *linenum*(4), and extensive symbolic information permit testing at the C *source* level. Every object file's symbol table is organized as shown below.

```
File name 1.
        Function 1.
                Local symbols for function 1.
        Function 2.
                Local symbols for function 2.
        ...
        Static externs for file 1.

File name 2.
        Function 1.
                Local symbols for function 1.
        Function 2.
                Local symbols for function 2.
        ...
        Static externs for file 2.
...

Defined global symbols.
Undefined global symbols.
```

The entry for a symbol is a fixed-length structure. The members of the structure hold the name (null padded), its value, and other information. The C structure is given below.

```
#define     SYMNMLEN     8
#define     FILNMLEN     14
#define     DIMNUM       4
```

```
struct  syment
{
 union                      /* all ways to get symbol name */
  {
    char       _n_name[SYMNMLEN]; /* symbol name */
    struct
    {
       long    _n_zeroes; /* == 0L when in string table */
       long    _n_offset; /* location of name in table */
    } _n_n;
    char       *_n_nptr[2]; /* allows overlaying */
  } _n;
  long         n_value;   /* value of symbol */
  short        n_scnum;   /* section number */
  unsigned short          n_type;/* type and derived type */
  char         n_sclass;  /* storage class */
  char         n_numaux;  /* number of aux entries */
};

#define       n_name        _n._n_name
#define       n_zeroes      _n._n_n._n_zeroes
#define       n_offset      _n._n_n._n_offset
#define       n_nptr        _n._n_nptr[1]
```

Meaningful values and explanations for them are given in both **syms.h** and *Common Object File Format*. Anyone who needs to interpret the entries should seek more information in these sources. Some symbols require more information than a single entry; they are followed by *auxiliary entries* that are the same size as a symbol entry. The format follows.

```
union auxent
{
   struct
   {
      long                      x_tagndx;
      union
      {
         struct
         {
               unsigned short   x_lnno;
               unsigned short   x_size;
         } x_lnsz;
         long x_fsize;
      } x_misc;
      union
      {
         struct
         {
               long             x_lnnoptr;
               long             x_endndx;
         }     x_fcn;
         struct
         {
               unsigned short   x_dimen[DIMNUM];
         }     x_ary;
      }        x_fcnary;
      unsigned short            x_tvndx;
   } x_sym;
   struct
   {
      char     x_fname[FILNMLEN];
   } x_file;
   struct
      {
      long     x_scnlen;
      unsigned short            x_nreloc;
      unsigned short            x_nlinno;
   } x_scn;
```

```
        struct
        {
           long                          x_tvfill;
           unsigned short                x_tvlen;
           unsigned short                x_tvran[2];
        } x_tv;
     };
```

Indexes of symbol table entries begin at *zero*.

SEE ALSO

sdb(1), a.out(4), linenum(4).

"Common Object File Format" in the *Programming Guide*.

WARNINGS

On machines on which **int**s are equivalent to **long**s, all **long**s have their type changed to **int.** Thus the information about which symbols are declared as **long**s and which, as **int**s, does not show up in the symbol table.

NAME

term − format of compiled term file.

SYNOPSIS

/usr/lib/terminfo/?/ *

DESCRIPTION

Compiled *terminfo*(4) descriptions are placed under the directory */usr/lib/terminfo*. In order to avoid a linear search of a huge UNIX system directory, a two-level scheme is used: */usr/lib/terminfo/c/name* where *name* is the name of the terminal, and *c* is the first character of *name*. Thus, **att4425** can be found in the file */usr/lib/terminfo/a/att4425*. Synonyms for the same terminal are implemented by multiple links to the same compiled file.

The format has been chosen so that it will be the same on all hardware. An 8-bit byte is assumed, but no assumptions about byte ordering or sign extension are made. Thus, these binary *terminfo*(4) files can be transported to other hardware with 8-bit bytes.

Short integers are stored in two 8-bit bytes. The first byte contains the least significant 8 bits of the value, and the second byte contains the most significant 8 bits. (Thus, the value represented is 256 * second + first.) The value **−1** is represented by **0377,0377**, and the value **−2** is represented by **0376,0377**; other negative values are illegal. Computers where this does not correspond to the hardware read the integers as two bytes and compute the result, making the compiled entries portable between machine types. The **−1** generally means that a capability is missing from this terminal. The **−2** means that the capability has been cancelled in the *terminfo*(4) source and also is to be considered missing.

The compiled file is created from the source file descriptions of the terminals (see the **−I** option of *infocmp*(1M)) by using the *terminfo*(4) compiler, *tic*(1M), and read by the routine **setupterm**(). (See *curses*(3X).) The file is divided into six parts: the header, terminal names, boolean flags, numbers, strings, and string table.

The header section begins the file. This section contains six short integers in the format described below. These integers are (1) the magic number (octal **0432**); (2) the size, in bytes, of the names section; (3) the number of bytes in the boolean section; (4) the number of short integers in the numbers section; (5) the number of offsets (short integers) in the strings section; (6) the size, in bytes, of the string table.

The terminal names section comes next. It contains the first line of the *terminfo*(4) description, listing the various names for the terminal, separated by the bar (¦) character (see *term*(5)). The section is terminated with an ASCII NUL character.

The boolean flags have one byte for each flag. This byte is either **0** or **1** as the flag is present or absent. The value of **2** means that the flag has been cancelled. The capabilities are in the same order as the file <**term.h**>.

Between the boolean section and the number section, a null byte will be inserted, if necessary, to ensure that the number section begins on an even byte. All short integers are aligned on a short word boundary.

The numbers section is similar to the boolean flags section. Each capability takes up two bytes, and is stored as a short integer. If the value represented is **−1** or **−2**, the capability is taken to be missing.

The strings section is also similar. Each capability is stored as a short integer, in the format above. A value of **−1** or **−2** means the capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table. Special characters in ^X or \c notation are stored in their interpreted form, not the printing representation. Padding information ($<nn>) and parameter information (%x) are stored intact in uninterpreted form.

The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each string is null terminated.

Note that it is possible for **setupterm**( ) to expect a different set of capabilities than are actually present in the file. Either the database may have been updated since **setupterm**( ) has been recompiled (resulting in extra unrecognized entries in the file) or the program may have been recompiled more recently than the database was updated (resulting in missing entries). The routine **setupterm**( ) must be prepared for both possibilities – this is why the numbers and sizes are included. Also, new capabilities must always be added at the end of the lists of boolean, number, and string capabilities. As an example, an octal dump of the description for the AT&T Model 37 KSR is included:

```
37|tty37|AT&T model 37 teletype,
    hc, os, xon,
    bel=^G, cr=\r, cub1=\b, cud1=\n, cuu1=\E7, hd=\E9,
    hu=\E8, ind=\n,
```

```
0000000 032 001      \0 032  \0 013  \0 021 001   3  \0   3   7   |   t
0000020   t   y   3   7   |   A   T   &   T       m   o   d   e   l    
0000040   3   7       t   e   l   e   t   y   p   e  \0  \0  \0  \0  \0
0000060  \0  \0  \0 001  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0  \0
0000100 001  \0  \0  \0  \0  \0 377 377 377 377 377 377 377 377 377 377
0000120 377 377 377 377 377 377 377 377 377 377 377 377 377 377   &  \0
0000140      \0 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0000160 377 377   "  \0 377 377 377 377   (  \0 377 377 377 377 377 377
0000200 377 377   0  \0 377 377 377 377 377 377 377 377   -  \0 377 377
0000220 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0000520 377 377 377 377 377 377 377 377 377 377 377 377 377 377   $  \0
0000540 377 377 377 377 377 377 377 377 377 377 377 377 377 377   *  \0
0000560 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0001160 377 377 377 377 377 377 377 377 377 377 377 377 377 377   3   7
0001200   |   t   t   y   3   7   |   A   T   &   T       m   o   d   e
0001220   l       3   7       t   e   l   e   t   y   p   e  \0  \r  \0
0001240  \n  \0  \n  \0 007  \0  \b  \0 033   8  \0 033   9  \0 033   7
0001260  \0  \0
0001261
```

99

Some limitations: total compiled entries cannot exceed 4096 bytes; all entries in the name field cannot exceed 128 bytes.

FILES

/usr/lib/terminfo/?/ *	compiled terminal description database
/usr/include/term.h	*terminfo*(4) header file

SEE ALSO

infocmp(1M), curses(3X), terminfo(4), term(5).

Chapter 10 of the *Programmer's Guide.*

NAME

terminfo – terminal capability data base

SYNOPSIS

/usr/lib/terminfo/?/ *

DESCRIPTION

terminfo is a compiled database (see *tic*(1M)) describing the capabilities of terminals. Terminals are described in *terminfo* source descriptions by giving a set of capabilities which they have, by describing how operations are performed, by describing padding requirements, and by specifying initialization sequences. This database is used by applications programs, such as *vi*(1) and *curses*(3X), so they can work with a variety of terminals without changes to the programs. To obtain the source description for a terminal, use the **−I** option of *infocmp*(1M).

Entries in *terminfo* source files consist of a number of comma-separated fields. White space after each comma is ignored. The first line of each terminal description in the *terminfo* database gives the name by which *terminfo* knows the terminal, separated by bar (|) characters. The first name given is the most common abbreviation for the terminal (this is the one to use to set the environment variable **TERM** in *$HOME/.profile*; see *profile*(4)), the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should contain no blanks and must be unique in the first 14 characters; the last name may contain blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, for example, for the AT&T 4425 terminal, **att4425**. Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. See *term*(5) for examples and more information on choosing names and synonyms.

CAPABILITIES

In the table below, the **Variable** is the name by which the C programmer (at the *terminfo* level) accesses the capability. The **Capname** is the short name for this variable used in the text of the database. It is used by a person updating the database and by the *tput*(1) command when asking what the value of the capability is for a particular terminal. The **Termcap Code** is a two-letter code that corresponds to the old *termcap* capability name.

Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64-1979 standard. Semantics are also intended to match those of the specification.

All string capabilities listed below may have padding specified, with the exception of those used for input. Input capabilities, listed under the **Strings** section in the table below, have names beginning with **key_**. The following indicators may appear at the end of the **Description** for a variable.

(G)	indicates that the string is passed through **tparm**( ) with parameters (parms) as given ($\#_i$).
(*)	indicates that padding may be based on the number of lines affected.
($\#_i$)	indicates the i^{th} parameter.

Variable	**Cap-name**	**Termcap Code**	**Description**
Booleans:			
auto_left_margin	bw	bw	**cub1** wraps from column 0 to last column
auto_right_margin	am	am	Terminal has automatic margins
no_esc_ctlc	xsb	xb	Beehive (f1 = escape, f2 = ctrl C)
ceol_standout_glitch	xhp	xs	Standout not erased by overwriting (hp)
eat_newline_glitch	xenl	xn	Newline ignored after 80 cols (*Concept*)

erase_overstrike	eo	eo	Can erase overstrikes with a blank
generic_type	gn	gn	Generic line type (e.g. dialup, switch).
hard_copy	hc	hc	Hardcopy terminal
hard_cursor	chts	HC	Cursor is hard to see.
has_meta_key	km	km	Has a meta key (shift, sets parity bit)
has_status_line	hs	hs	Has extra "status line"
insert_null_glitch	in	in	Insert mode distinguishes nulls
memory_above	da	da	Display may be retained above the screen
memory_below	db	db	Display may be retained below the screen
move_insert_mode	mir	mi	Safe to move while in insert mode
move_standout_mode	msgr	ms	Safe to move in standout modes
needs_xon_xoff	nxon	nx	Padding won't work, xon/xoff required
non_rev_rmcup	nrrmc	NR	**smcup** does not reverse **rmcup**
no_pad_char	npc	NP	Pad character doesn't exist
over_strike	os	os	Terminal overstrikes on hard-copy terminal
prtr_silent	mc5i	5i	Printer won't echo on screen.
status_line_esc_ok	eslok	es	Escape can be used on the status line
dest_tabs_magic_smso	xt	xt	Destructive tabs, magic **smso** char (t1061)
tilde_glitch	hz	hz	Hazeltine; can't print tildes(˜)
transparent_underline	ul	ul	Underline character overstrikes
xon_xoff	xon	xo	Terminal uses xon/xoff handshaking

Numbers:

columns	cols	co	Number of columns in a line
init_tabs	it	it	Tabs initially every # spaces.
label_height	lh	lh	Number of rows in each label
label_width	lw	lw	Number of cols in each label
lines	lines	li	Number of lines on screen or page
lines_of_memory	lm	lm	Lines of memory if > **lines**; **0** means varies
magic_cookie_glitch	xmc	sg	Number blank chars left by **smso** or **rmso**
num_labels	nlab	Nl	Number of labels on screen (start at 1)
padding_baud_rate	pb	pb	Lowest baud rate where padding needed
virtual_terminal	vt	vt	Virtual terminal number (UNIX system)
width_status_line	wsl	ws	Number of columns in status line

Strings:

acs_chars	acsc	ac	Graphic charset pairs aAbBcC - def=vt100+
back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
carriage_return	cr	cr	Carriage return (*)
change_scroll_region	csr	cs	Change to lines #1 thru #2 (vt100) (G)
char_padding	rmp	rP	Like **ip** but when in replace mode
clear_all_tabs	tbc	ct	Clear all tab stops
clear_margins	mgc	MC	Clear left and right soft margins
clear_screen	clear	cl	Clear screen and home cursor (*)
clr_bol	el1	cb	Clear to beginning of line, inclusive
clr_eol	el	ce	Clear to end of line
clr_eos	ed	cd	Clear to end of display (*)
column_address	hpa	ch	Horizontal position absolute (G)
command_character	cmdch	CC	Term. settable cmd char in prototype
cursor_address	cup	cm	Cursor motion to row #1 col #2 (G)
cursor_down	cud1	do	Down one line
cursor_home	home	ho	Home cursor (if no **cup**)
cursor_invisible	civis	vi	Make cursor invisible
cursor_left	cub1	le	Move cursor left one space.
cursor_mem_address	mrcup	CM	Memory relative cursor addressing (G)
cursor_normal	cnorm	ve	Make cursor appear normal (undo **vs/vi**)
cursor_right	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll	ll	ll	Last line, first column (if no **cup**)
cursor_up	cuu1	up	Upline (cursor up)
cursor_visible	cvvis	vs	Make cursor very visible
delete_character	dch1	dc	Delete character (*)
delete_line	dl1	dl	Delete line (*)
dis_status_line	dsl	ds	Disable status line
down_half_line	hd	hd	Half-line down (forward 1/2 linefeed)
ena_acs	enacs	eA	Enable alternate char set
enter_alt_charset_mode	smacs	as	Start alternate character set
enter_am_mode	smam	SA	Turn on automatic margins
enter_blink_mode	blink	mb	Turn on blinking
enter_bold_mode	bold	md	Turn on bold (extra bright) mode
enter_ca_mode	smcup	ti	String to begin programs that use **cup**

enter_delete_mode	smdc	dm	Delete mode (enter)
enter_dim_mode	dim	mh	Turn on half-bright mode
enter_insert_mode	smir	im	Insert mode (enter);
enter_protected_mode	prot	mp	Turn on protected mode
enter_reverse_mode	rev	mr	Turn on reverse video mode
enter_secure_mode	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode	smso	so	Begin standout mode
enter_underline_mode	smul	us	Start underscore mode
enter_xon_mode	smxon	SX	Turn on xon/xoff handshaking
erase_chars	ech	ec	Erase #1 characters (G)
exit_alt_charset_mode	rmacs	ae	End alternate character set
exit_am_mode	rmam	RA	Turn off automatic margins
exit_attribute_mode	sgr0	me	Turn off all attributes
exit_ca_mode	rmcup	te	String to end programs that use **cup**
exit_delete_mode	rmdc	ed	End delete mode
exit_insert_mode	rmir	ei	End insert mode;
exit_standout_mode	rmso	se	End standout mode
exit_underline_mode	rmul	ue	End underscore mode
exit_xon_mode	rmxon	RX	Turn off xon/xoff handshaking
flash_screen	flash	vb	Visible bell (may not move cursor)
form_feed	ff	ff	Hardcopy terminal page eject (*)
from_status_line	fsl	fs	Return from status line
init_1string	is1	i1	Terminal initialization string
init_2string	is2	is	Terminal initialization string
init_3string	is3	i3	Terminal initialization string
init_file	if	if	Name of initialization file containing **is**
init_prog	iprog	iP	Path name of program for init.
insert_character	ich1	ic	Insert character
insert_line	il1	al	Add new blank line (*)
insert_padding	ip	ip	Insert pad after character inserted (*)
key_a1	ka1	K1	KEY_A1, 0534, Upper left of keypad
key_a3	ka3	K3	KEY_A3, 0535, Upper right of keypad
key_b2	kb2	K2	KEY_B2, 0536, Center of keypad
key_backspace	kbs	kb	KEY_BACKSPACE, 0407, Sent by backspace key
key_beg	kbeg	@1	KEY_BEG, 0542, Sent by beg(inning) key
key_btab	kcbt	kB	KEY_BTAB, 0541, Sent by back-tab key

key_c1	kc1	K4	KEY_C1, 0537, Lower left of keypad
key_c3	kc3	K5	KEY_C3, 0540, Lower right of keypad
key_cancel	kcan	@2	KEY_CANCEL, 0543, Sent by cancel key
key_catab	ktbc	ka	KEY_CATAB, 0526, Sent by clear-all-tabs key
key_clear	kclr	kC	KEY_CLEAR, 0515, Sent by clear-screen or erase key
key_close	kclo	@3	KEY_CLOSE, 0544, Sent by close key
key_command	kcmd	@4	KEY_COMMAND, 0545, Sent by cmd (command) key
key_copy	kcpy	@5	KEY_COPY, 0546, Sent by copy key
key_create	kcrt	@6	KEY_CREATE, 0547, Sent by create key
key_ctab	kctab	kt	KEY_CTAB, 0525, Sent by clear-tab key
key_dc	kdch1	kD	KEY_DC, 0512, Sent by delete-character key
key_dl	kdl1	kL	KEY_DL, 0510, Sent by delete-line key
key_down	kcud1	kd	KEY_DOWN, 0402, Sent by terminal down-arrow key
key_eic	krmir	kM	KEY_EIC, 0514, Sent by **rmir** or **smir** in insert mode
key_end	kend	@7	KEY_END, 0550, Sent by end key
key_enter	kent	@8	KEY_ENTER, 0527, Sent by enter/send key
key_eol	kel	kE	KEY_EOL, 0517, Sent by clear-to-end-of-line key
key_eos	ked	kS	KEY_EOS, 0516, Sent by clear-to-end-of-screen key
key_exit	kext	@9	KEY_EXIT, 0551, Sent by exit key
key_f0	kf0	k0	KEY_F(0), 0410, Sent by function key f0
key_f1	kf1	k1	KEY_F(1), 0411, Sent by function key f1
key_f2	kf2	k2	KEY_F(2), 0412, Sent by function key f2
key_f3	kf3	k3	KEY_F(3), 0413, Sent by function key f3
key_f4	kf4	k4	KEY_F(4), 0414, Sent by function key f4
key_f5	kf5	k5	KEY_F(5), 0415, Sent by function key f5
key_f6	kf6	k6	KEY_F(6), 0416, Sent by function key f6
key_f7	kf7	k7	KEY_F(7), 0417, Sent by function key f7
key_f8	kf8	k8	KEY_F(8), 0420, Sent by function key f8
key_f9	kf9	k9	KEY_F(9), 0421, Sent by function key f9
key_f10	kf10	k;	KEY_F(10), 0422, Sent by function key f10
key_f11	kf11	F1	KEY_F(11), 0423, Sent by function key f11
key_f12	kf12	F2	KEY_F(12), 0424, Sent by function key f12
key_f13	kf13	F3	KEY_F(13), 0425, Sent by function key f13
key_f14	kf14	F4	KEY_F(14), 0426, Sent by function key f14
key_f15	kf15	F5	KEY_F(15), 0427, Sent by function key f15
key_f16	kf16	F6	KEY_F(16), 0430, Sent by function key f16

key_f17	kf17	F7	KEY_F(17), 0431, Sent by function key f17
key_f18	kf18	F8	KEY_F(18), 0432, Sent by function key f18
key_f19	kf19	F9	KEY_F(19), 0433, Sent by function key f19
key_f20	kf20	FA	KEY_F(20), 0434, Sent by function key f20
key_f21	kf21	FB	KEY_F(21), 0435, Sent by function key f21
key_f22	kf22	FC	KEY_F(22), 0436, Sent by function key f22
key_f23	kf23	FD	KEY_F(23), 0437, Sent by function key f23
key_f24	kf24	FE	KEY_F(24), 0440, Sent by function key f24
key_f25	kf25	FF	KEY_F(25), 0441, Sent by function key f25
key_f26	kf26	FG	KEY_F(26), 0442, Sent by function key f26
key_f27	kf27	FH	KEY_F(27), 0443, Sent by function key f27
key_f28	kf28	FI	KEY_F(28), 0444, Sent by function key f28
key_f29	kf29	FJ	KEY_F(29), 0445, Sent by function key f29
key_f30	kf30	FK	KEY_F(30), 0446, Sent by function key f30
key_f31	kf31	FL	KEY_F(31), 0447, Sent by function key f31
key_f32	kf32	FM	KEY_F(32), 0450, Sent by function key f32
key_f33	kf33	FN	KEY_F(13), 0451, Sent by function key f13
key_f34	kf34	FO	KEY_F(34), 0452, Sent by function key f34
key_f35	kf35	FP	KEY_F(35), 0453, Sent by function key f35
key_f36	kf36	FQ	KEY_F(36), 0454, Sent by function key f36
key_f37	kf37	FR	KEY_F(37), 0455, Sent by function key f37
key_f38	kf38	FS	KEY_F(38), 0456, Sent by function key f38
key_f39	kf39	FT	KEY_F(39), 0457, Sent by function key f39
key_f40	kf40	FU	KEY_F(40), 0460, Sent by function key f40
key_f41	kf41	FV	KEY_F(41), 0461, Sent by function key f41
key_f42	kf42	FW	KEY_F(42), 0462, Sent by function key f42
key_f43	kf43	FX	KEY_F(43), 0463, Sent by function key f43
key_f44	kf44	FY	KEY_F(44), 0464, Sent by function key f44
key_f45	kf45	FZ	KEY_F(45), 0465, Sent by function key f45
key_f46	kf46	Fa	KEY_F(46), 0466, Sent by function key f46
key_f47	kf47	Fb	KEY_F(47), 0467, Sent by function key f47
key_f48	kf48	Fc	KEY_F(48), 0470, Sent by function key f48
key_f49	kf49	Fd	KEY_F(49), 0471, Sent by function key f49
key_f50	kf50	Fe	KEY_F(50), 0472, Sent by function key f50
key_f51	kf51	Ff	KEY_F(51), 0473, Sent by function key f51
key_f52	kf52	Fg	KEY_F(52), 0474, Sent by function key f52

key_f53	kf53	Fh	KEY_F(53), 0475, Sent by function key f53
key_f54	kf54	Fi	KEY_F(54), 0476, Sent by function key f54
key_f55	kf55	Fj	KEY_F(55), 0477, Sent by function key f55
key_f56	kf56	Fk	KEY_F(56), 0500, Sent by function key f56
key_f57	kf57	Fl	KEY_F(57), 0501, Sent by function key f57
key_f58	kf58	Fm	KEY_F(58), 0502, Sent by function key f58
key_f59	kf59	Fn	KEY_F(59), 0503, Sent by function key f59
key_f60	kf60	Fo	KEY_F(60), 0504, Sent by function key f60
key_f61	kf61	Fp	KEY_F(61), 0505, Sent by function key f61
key_f62	kf62	Fq	KEY_F(62), 0506, Sent by function key f62
key_f63	kf63	Fr	KEY_F(63), 0507, Sent by function key f63
key_find	kfnd	@0	KEY_FIND, 0552, Sent by find key
key_help	khlp	%1	KEY_HELP, 0553, Sent by help key
key_home	khome	kh	KEY_HOME, 0406, Sent by home key
key_ic	kich1	kI	KEY_IC, 0513, Sent by ins-char/enter ins-mode key
key_il	kil1	kA	KEY_IL, 0511, Sent by insert-line key
key_left	kcub1	kl	KEY_LEFT, 0404, Sent by terminal left-arrow key
key_ll	kll	kH	KEY_LL, 0533, Sent by home-down key
key_mark	kmrk	%2	KEY_MARK, 0554, Sent by mark key
key_message	kmsg	%3	KEY_MESSAGE, 0555, Sent by message key
key_move	kmov	%4	KEY_MOVE, 0556, Sent by move key
key_next	knxt	%5	KEY_NEXT, 0557, Sent by next-object key
key_npage	knp	kN	KEY_NPAGE, 0522, Sent by next-page key
key_open	kopn	%6	KEY_OPEN, 0560, Sent by open key
key_options	kopt	%7	KEY_OPTIONS, 0561, Sent by options key
key_ppage	kpp	kP	KEY_PPAGE, 0523, Sent by previous-page key
key_previous	kprv	%8	KEY_PREVIOUS, 0562, Sent by previous-object key
key_print	kprt	%9	KEY_PRINT, 0532, Sent by print or copy key
key_redo	krdo	%0	KEY_REDO, 0563, Sent by redo key
key_reference	kref	&1	KEY_REFERENCE, 0564, Sent by ref(erence) key
key_refresh	krfr	&2	KEY_REFRESH, 0565, Sent by refresh key
key_replace	krpl	&3	KEY_REPLACE, 0566, Sent by replace key
key_restart	krst	&4	KEY_RESTART, 0567, Sent by restart key
key_resume	kres	&5	KEY_RESUME, 0570, Sent by resume key
key_right	kcuf1	kr	KEY_RIGHT, 0405, Sent by terminal right-arrow key
key_save	ksav	&6	KEY_SAVE, 0571, Sent by save key

key_sbeg	kBEG	&9	KEY_SBEG, 0572, Sent by shifted beginning key
key_scancel	kCAN	&0	KEY_SCANCEL, 0573, Sent by shifted cancel key
key_scommand	kCMD	*1	KEY_SCOMMAND, 0574, Sent by shifted command key
key_scopy	kCPY	*2	KEY_SCOPY, 0575, Sent by shifted copy key
key_screate	kCRT	*3	KEY_SCREATE, 0576, Sent by shifted create key
key_sdc	kDC	*4	KEY_SDC, 0577, Sent by shifted delete-char key
key_sdl	kDL	*5	KEY_SDL, 0600, Sent by shifted delete-line key
key_select	kslt	*6	KEY_SELECT, 0601, Sent by select key
key_send	kEND	*7	KEY_SEND, 0602, Sent by shifted end key
key_seol	kEOL	*8	KEY_SEOL, 0603, Sent by shifted clear-line key
key_sexit	kEXT	*9	KEY_SEXIT, 0604, Sent by shifted exit key
key_sf	kind	kF	KEY_SF, 0520, Sent by scroll-forward/down key
key_sfind	kFND	*0	KEY_SFIND, 0605, Sent by shifted find key
key_shelp	kHLP	#1	KEY_SHELP, 0606, Sent by shifted help key
key_shome	kHOM	#2	KEY_SHOME, 0607, Sent by shifted home key
key_sic	kIC	#3	KEY_SIC, 0610, Sent by shifted input key
key_sleft	kLFT	#4	KEY_SLEFT, 0611, Sent by shifted left-arrow key
key_smessage	kMSG	%a	KEY_SMESSAGE, 0612, Sent by shifted message key
key_smove	kMOV	%b	KEY_SMOVE, 0613, Sent by shifted move key
key_snext	kNXT	%c	KEY_SNEXT, 0614, Sent by shifted next key
key_soptions	kOPT	%d	KEY_SOPTIONS, 0615, Sent by shifted options key
key_sprevious	kPRV	%e	KEY_SPREVIOUS, 0616, Sent by shifted prev key
key_sprint	kPRT	%f	KEY_SPRINT, 0617, Sent by shifted print key
key_sr	kri	kR	KEY_SR, 0521, Sent by scroll-backward/up key
key_sredo	kRDO	%g	KEY_SREDO, 0620, Sent by shifted redo key
key_sreplace	kRPL	%h	KEY_SREPLACE, 0621, Sent by shifted replace key
key_sright	kRIT	%i	KEY_SRIGHT, 0622, Sent by shifted right-arrow key
key_srsume	kRES	%j	KEY_SRSUME, 0623, Sent by shifted resume key
key_ssave	kSAV	!1	KEY_SSAVE, 0624, Sent by shifted save key
key_ssuspend	kSPD	!2	KEY_SSUSPEND, 0625, Sent by shifted suspend key
key_stab	khts	kT	KEY_STAB, 0524, Sent by set-tab key
key_sundo	kUND	!3	KEY_SUNDO, 0626, Sent by shifted undo key
key_suspend	kspd	&7	KEY_SUSPEND, 0627, Sent by suspend key
key_undo	kund	&8	KEY_UNDO, 0630, Sent by undo key
key_up	kcuu1	ku	KEY_UP, 0403, Sent by terminal up-arrow key
keypad_local	rmkx	ke	Out of "keypad-transmit" mode

keypad_xmit	smkx	ks	Put terminal in "keypad-transmit" mode
lab_f0	lf0	l0	Labels on function key f0 if not f0
lab_f1	lf1	l1	Labels on function key f1 if not f1
lab_f2	lf2	l2	Labels on function key f2 if not f2
lab_f3	lf3	l3	Labels on function key f3 if not f3
lab_f4	lf4	l4	Labels on function key f4 if not f4
lab_f5	lf5	l5	Labels on function key f5 if not f5
lab_f6	lf6	l6	Labels on function key f6 if not f6
lab_f7	lf7	l7	Labels on function key f7 if not f7
lab_f8	lf8	l8	Labels on function key f8 if not f8
lab_f9	lf9	l9	Labels on function key f9 if not f9
lab_f10	lf10	la	Labels on function key f10 if not f10
label_off	rmln	LF	Turn off soft labels
label_on	smln	LO	Turn on soft labels
meta_off	rmm	mo	Turn off "meta mode"
meta_on	smm	mm	Turn on "meta mode" (8th bit)
newline	nel	nw	Newline (behaves like **cr** followed by **lf**)
pad_char	pad	pc	Pad character (rather than null)
parm_dch	dch	DC	Delete #1 chars (G *)
parm_delete_line	dl	DL	Delete #1 lines (G *)
parm_down_cursor	cud	DO	Move cursor down #1 lines. (G *)
parm_ich	ich	IC	Insert #1 blank chars (G *)
parm_index	indn	SF	Scroll forward #1 lines. (G)
parm_insert_line	il	AL	Add #1 new blank lines (G *)
parm_left_cursor	cub	LE	Move cursor left #1 spaces (G)
parm_right_cursor	cuf	RI	Move cursor right #1 spaces. (G *)
parm_rindex	rin	SR	Scroll backward #1 lines. (G)
parm_up_cursor	cuu	UP	Move cursor up #1 lines. (G *)
pkey_key	pfkey	pk	Prog funct key #1 to type string #2
pkey_local	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit	pfx	px	Prog funct key #1 to xmit string #2
plab_norm	pln	pn	Prog label #1 to show string #2
print_screen	mc0	ps	Print contents of the screen
prtr_non	mc5p	pO	Turn on the printer for #1 bytes
prtr_off	mc4	pf	Turn off the printer
prtr_on	mc5	po	Turn on the printer

repeat_char	rep	rp	Repeat char #1 #2 times (G *)
req_for_input	rfi	RF	Send next input char (for ptys)
reset_1string	rs1	r1	Reset terminal completely to sane modes
reset_2string	rs2	r2	Reset terminal completely to sane modes
reset_3string	rs3	r3	Reset terminal completely to sane modes
reset_file	rf	rf	Name of file containing reset string
restore_cursor	rc	rc	Restore cursor to position of last sc
row_address	vpa	cv	Vertical position absolute (G)
save_cursor	sc	sc	Save cursor position.
scroll_forward	ind	sf	Scroll text up
scroll_reverse	ri	sr	Scroll text down
set_attributes	sgr	sa	Define the video attributes #1-#9 (G)
set_left_margin	smgl	ML	Set soft left margin
set_right_margin	smgr	MR	Set soft right margin
set_tab	hts	st	Set a tab in all rows, current column.
set_window	wind	wi	Current window is lines #1-#2 cols #3-#4 (G)
tab	ht	ta	Tab to next 8 space hardware tab stop.
to_status_line	tsl	ts	Go to status line, col #1 (G)
underline_char	uc	uc	Underscore one char and move past it
up_half_line	hu	hu	Half-line up (reverse 1/2 linefeed)
xoff_character	xoffc	XF	X-off character
xon_character	xonc	XN	X-on character

SAMPLE ENTRY

The following entry, which describes the *Concept* − 100 terminal, is among the more complex entries in the *terminfo* file as of this writing.

```
concept100|c100|concept|c104|c100-4p|concept 100,
 am, db, eo, in, mir, ul, xenl,
 cols#80, lines#24, pb#9600, vt#8,
 bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>,
 cnorm=\Ew, cr=^M$<9>, cub1=^H, cud1=^J,
 cuf1=\E=, cup=\Ea%p1%' '%+%c%p2%' '%+%c,
 cuu1=\E;, cvvis=\EW, dch1=\E^A$<16*>, dim=\EE,
 dl1=\E^B$<3*>, ed=\E^C$<16*>, el=\E^U$<16>,
 flash=\Ek$<20>\EK, ht=\t$<8>, il1=\E^R$<3*>,
 ind=^J, .ind=^J$<9>, ip=$<16*>,
 is2=\EU\Ef\E7\E5\E8\El\ENH\EK\E\0\Eo&\0\Eo\47\E,
 kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
 kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
 prot=\EI, rep=\Er%p1%c%p2%' '%+%c$<.2*>,
 rev=\ED, rmcup=\Ev\s\s\s\s$<6>\Ep\r\n,
 rmir=\E\0, rmkx=\Ex, rmso=\Ed\Ee, rmul=\Eg,
 rmul=\Eg, sgr0=\EN\0, smcup=\EU\Ev\s\s8p\Ep\r,
 smir=\E^P, smkx=\EX, smso=\EE\ED, smul=\EG,
```

112

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Lines beginning with "#" are taken as comment lines. Capabilities in *terminfo* are of three types: boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or particular features, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For instance, the fact that the *Concept* has *automatic margins* (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the *Concept* includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **cols**, which indicates the number of columns the terminal has, gives the value **80** for the *Concept*. The value may be specified in decimal, octal or hexadecimal using normal C conventions.

Finally, string-valued capabilities, such as **el** (clear to end of line sequence) are given by the two- to five-character capname, an '=', and then a string ending at the next following comma. A delay in milliseconds may appear anywhere in such a capability, enclosed in **$<..>** brackets, as in **el=\EK$<3>**, and padding characters are supplied by **tputs**( ) (see *curses*(3X)) to provide this delay. The delay can be either a number, e.g., **20**, or a number followed by an '*' (i.e., **3***), a '/' (i.e., **5/**), or both (i.e., **10*/**). A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of lines affected. This is always one unless the terminal has **in** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form **3.5** to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.) A '/' indicates that the padding is mandatory. Otherwise, if the terminal has **xon** defined, the padding information is advisory and will only be used for cost estimates or when the terminal is in raw mode. Mandatory padding will be transmitted regardless of the setting of **xon**.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. Both **\E** and **\e** map to an ESCAPE character, ^*x* maps to a control−*x* for any appropriate *x*, and the sequences **\n, \l, \r, \t, \b, \f,** and **\s** give a newline, linefeed, return, tab, backspace, formfeed, and space, respectively. Other escapes include: **\^** for caret (^); \\ for backslash (\); **\,** for comma (,); **\:** for colon (:); and **\0** for null. (**\0** will actually produce **\200**, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a backslash (e.g., \123).

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above. Note that capabilities are defined in a left-to-right order and, therefore, a prior definition will override a later definition.

Preparing Descriptions

The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with **vi**(1) to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the **terminfo** file to describe it or the inability of **vi**(1) to work with that terminal. To test a new terminal description, set the environment variable **TERMINFO** to a pathname of a directory containing the compiled description you are working on and programs will look there rather than in */usr/lib/terminfo*. To get the padding for insert-line correct (if the terminal manufacturer did not document it) a severe test is to comment out **xon**, edit a large file at 9600 baud with **vi**(1), delete 16 or so lines from the middle of the screen, then hit the **u** key several times quickly. If the display is corrupted, more padding is usually needed. A similar test can be used for insert-character.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal has a screen, then the number of lines on the screen is given by the **lines** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (**os** applies to storage scope terminals, such as Tektronix 4010 series, as well as hard-copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc) give this as **bel**. If the terminal uses the xon-xoff flow-control protocol, like most terminals, specify **xon**.

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1**. Similarly, codes to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**. These local cursor motions should not alter the text they pass over; for example, you would not normally use "**cuf1**=\s" because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a screen terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and should never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and screen terminals. Thus the model 33 teletype is described as

```
33 | tty33 | tty | model 33 teletype,
   bel=^G, cols#72, cr=^M, cud1=^J, hc, ind=^J, os,
```

while the Lear Siegler ADM−3 is described as

```
adm3 | 3 | lsi adm3,
   am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H,
   cud1=^J, ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with **printf**(3S)-like escapes (%**x**) in it. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The parameter mechanism uses a stack and special % codes to manipulate it in the manner of a Reverse Polish Notation (postfix) calculator. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary. Binary operations are in postfix form with the operands in the usual order. That is, to get x−5 one would use %**gx**%{**5**}%−.

The % encodings have the following meanings:

%% outputs '%'

%[[:]*flags*][*width*[.*precision*]][**doxXs**]
as in printf, flags are [−+#] and space

%c	print pop() gives %c
%p[1−9]	push i^{th} parm
%P[a−z]	set variable [a−z] to pop()
%g[a−z]	get variable [a−z] and push it
%'c'	push char constant *c*
%{*nn*}	push decimal constant *nn*
%l	push strlen(pop())
%+ %− %* %/ %m	arithmetic (%m is mod): push(pop() op pop())
%& %| %^	bit operations: push(pop() op pop())
%= %> %<	logical operations: push(pop() op pop())
%A %O	logical operations: and, or
%! %~	unary operations: push(op pop())
%i	(for ANSI terminals) add 1 to first parm, if one parm present, or first two parms, if more than one parm present
%? expr %t thenpart %e elsepart %;	if-then-else, %e elsepart is optional; else-if's are possible ala Algol 68: %? c_1 %t b_1 %e c_2 %t b_2 %e c_3 %t b_3 %e c_4 %t b_4 %e b_5%; c_i are conditions, b_i are bodies.

If the "−" flag is used with "%[doxXs]", then a colon (:) must be placed between the "%" and the "−" to differentiate the flag from the binary "%−" operator, .e.g "%:−16.16s".

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent **\E&a12c03Y** padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are zero-padded as two digits. Thus its **cup** capability is "**cup**=\E&a%p2%2.2dc%p1%2.2dY$<6>".

The Micro-Term ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary, "**cup**=^T%p1%c%p2%c". Terminals which use "%c" need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **\n**, **^D**, and **\r**, as the system may change or discard them. (The library routines dealing with *terminfo* set tty modes so that tabs are never expanded, so **\t** is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "**cup**=\E=%p1%'\s'%+%c%p2%'\s'%+%c". After sending "\E=", this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values), and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the **\EH** sequence on Hewlett-Packard terminals cannot be used for **home** without losing some of the other features on the terminal.)

If the terminal has row or column absolute-cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two-parameter sequence (as with the Hewlett-Packard 2645) and can be used in preference to **cup**. If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup**, such as the Tektronix 4025.

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the beginning of the line to the current position inclusive, leaving the cursor where it is, this should be given as **el1**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl**.

If the terminal has a settable destructive scrolling region (like the VT100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command -- the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

To determine whether a terminal has destructive scrolling regions or non-destructive scrolling regions, create a scrolling region in the middle of the screen, place data on the bottom line of the scrolling region, move the cursor to the top line of the scrolling region, and do a reverse index (**ri**) followed by a delete line (**dl1**) or index (**ind**). If the data that was originally on the bottom line of the scrolling region was restored into the scrolling region by the **dl1** or **ind**, then the terminal has non-destructive scrolling regions. Otherwise, it has destructive scrolling regions. Do not specify **csr** if the terminal has non-destructive scrolling regions, unless **ind**, **ri**, **indn**, **rin**, **dl**, and **dl1** all simulate destructive scrolling.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling a full screen may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character operations which can be described using *terminfo*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the *Concept* 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "**abc def**" using local cursor motions (not spaces) between the **abc** and the

def. Then position the cursor before the **abc** and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the **abc** shifts over to the **def** which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". While these are two logically separate attributes (one line versus multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

terminfo can describe both terminals which have an insert mode and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post-insert padding is needed, give this as a number of milliseconds padding in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir**/**rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ich1** *n* times.

If padding is necessary between characters typed while not in insert mode, give this as a number of milliseconds padding in **rmp**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n*, to delete *n* characters, and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode* (see *curses*(3X)), representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If you have a choice, reverse-video plus half-bright is good, or reverse-video alone; however, different users have different preferences on different terminals.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Micro-Term MIME, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking), **bold** (bold or extra-bright), **dim** (dim or half-bright), **invis** (blanking or invisible text), **prot** (protected), **rev** (reverse-video), **sgr0** (turn off all attribute modes), **smacs** (enter alternate-character-set mode), and **rmacs** (exit alternate-character-set mode). Turning on any of these modes singly may or may not turn off other modes. If a command is necessary before alternate character set mode is entered, give the sequence in **enacs** (enable alternate-character-set mode).

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking nine parameters. Each parameter is either **0** or non-zero, as the corresponding attribute is on or off. The nine parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist. (See the example at the end of this section.)

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), then this can be given as **flash**; it must not move the cursor. A good flash can be done by changing the screen into reverse video, pad for 200 ms, then return the screen to normal video.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. The boolean **chts** should also be given. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given which undoes the effects of either of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals like the *Concept* with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the Tektronix 4025, where **smcup** sets the command character to be the one used by **terminfo**. If the **smcup** sequence will not restore the screen after an **rmcup** sequence is output (to the state prior to outputting **rmcup**), specify **nrrmc**.

If your terminal generates underlined characters by using the underline character (with no special codes needed) even though it does not otherwise overstrike characters, then you should give the capability **ul**. For terminals where a character overstriking another leaves both characters on the screen, give the capability **os**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Example of highlighting: assume that the terminal under question needs the following escape sequences to turn on various modes.

tparm parameter	attribute	escape sequence
	none	\E[0m
p1	standout	\E[0;4;7m
p2	underline	\E[0;3m
p3	reverse	\E[0;4m

p4	blink	\E[0;5m
p5	dim	\E[0;7m
p6	bold	\E[0;3;4m
p7	invis	\E[0;8m
p8	protect	not available
p9	altcharset	^O (off) ^N(on)

Note that each escape sequence requires a **0** to turn off other modes before turning on its own mode. Also note that, as suggested above, *standout* is set up to be the combination of *reverse* and *dim*. Also, since this terminal has no *bold* mode, *bold* is set up as the combination of *reverse* and *underline*. In addition, to allow combinations, such as *underline+blink*, the sequence to use would be **\E[0;3;5m**. The terminal doesn't have *protect* mode, either, but that cannot be simulated in any way, so **p8** is ignored. The *altcharset* mode is different in that it is either **^O** or **^N** depending on whether it is off or on. If all modes were to be turned on, the sequence would be **\E[0;3;4;5;7;8m^N**.

Now look at when different sequences are output. For example, **;3** is output when either **p2** or **p6** is true, that is, if either *underline* or *bold* modes are turned on. Writing out the above sequences, along with their dependencies, gives the following:

sequence	when to output	terminfo translation
\E[0	always	\E[0
;3	if p2 or p6	%?%p2%p6%|%t;3%;
;4	if p1 or p3 or p6	%?%p1%p3%|%p6%|%t;4%;
;5	if p4	%?%p4%t;5%;
;7	if p1 or p5	%?%p1%p5%|%t;7%;
;8	if p7	%?%p7%t;8%;
m	always	m
^N or ^O	if p9 ^N, else ^O	%?%p9%t^N%e^O%;

Putting this all together into the **sgr** sequence gives:

```
sgr=\E[0%?%p2%p6%|%t;3%;%?%p1%p3%|%p6%
    |%t;4%;%?%p5%t;5%;%?%p1%p5%
    |%t;7%;%?%p7%t;8%;m%?%p9%t^N%e^O%;,
```

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. Otherwise the keypad is assumed to always transmit.

The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome** respectively. If there are function keys such as f0, f1, ..., f63, the codes they send can be given as **kf0, kf1, ..., kf63**. If the first 11 keys have labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10**. The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1**, **ka3**, **kb2**, **kc1**, and **kc3**. These keys are useful when the effects of a 3 by 3 directional pad are needed. Further keys are defined above in the capabilities list.

Strings to program function keys can be given as **pfkey**, **pfloc**, and **pfx**. A string to program their soft-screen labels can be given as **pln**. Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal-dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local mode; and **pfx** causes the string to be transmitted to the computer. The capabilities **nlab**, **lw** and **lh** define how many soft labels there are and their width and height. If there are commands to turn the labels on and off, give them in **smln** and **rmln**. **smln** is normally output after one or more **pln** sequences to make sure that the change becomes visible.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt**. By convention, if the teletype modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by **tput init** (see *tput*(1)) to determine whether to set the mode for hardware tab expansion and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the *terminfo* description can assume that they are properly set. If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row).

Other capabilities include: **is1**, **is2**, and **is3**, initialization strings for the terminal; **iprog**, the path name of a program to be run to initialize the terminal; and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the *terminfo* description. They must be sent to the terminal each time the user logs in and be output in the following order: run the program **iprog**; output **is1**; output **is2**; set the margins using **mgc**, **smgl** and **smgr**; set the tabs using **tbc** and **hts**; print the file **if**; and finally output **is3**. This is usually done using the **init** option of *tput*(1); see *profile*(4).

Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. Sequences that do a harder reset from a totally unknown state can be given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is1**, **is2**, **is3**, and **if**. (The method using files, **if** and **rf**, is used for a few terminals, from */usr/lib/tabset/* * ; however, the recommended method is to use the initialization and reset strings.) These strings are output by **tput reset**, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs1**, **rs2**, **rs3**, and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set a terminal into 80-column mode would normally be part of **is2**, but on some terminals it causes an annoying glitch on the screen and is not normally needed since the terminal is usually already in 80-column mode.

If a more complex sequence is needed to set the tabs than can be described by using **tbc** and **hts**, the sequence can be placed in **is2** or **if**.

If there are commands to set and clear margins, they can be given as **mgc** (clear all margins), **smgl** (set left margin), and **smgr** (set right margin).

Delays

Certain capabilities control padding in the *tty*(7) driver. These are primarily needed by hard-copy terminals, and are used by **tput init** to set tty modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** can be used to set the appropriate delay bits to be set in the tty driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Status Lines

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit h19's 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings that go to a given column of the status line and return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The capability **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to.

If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

Line Graphics

If the terminal has a line drawing alternate character set, the mapping of glyph to character would be given in **acsc**. The definition of this string is based on the alternate character set used in the DEC VT100 terminal, extended slightly with some characters from the AT&T 4410v1 terminal.

glyph name	vt100+ character
arrow pointing right	+
arrow pointing left	,
arrow pointing down	.
solid square block	0
lantern symbol	I
arrow pointing up	–
diamond	‘
checker board (stipple)	a
degree symbol	f
plus/minus	g
board of squares	h
lower right corner	j
upper right corner	k
upper left corner	l
lower left corner	m
plus	n
scan line 1	o
horizontal line	q
scan line 9	s
left tee (|–)	t
right tee (–|)	u
bottom tee (⊥)	v
top tee (⊤)	w
vertical line	x
bullet	~

The best way to describe a new terminal's line graphics set is to add a third column to the above table with the characters for the new terminal that produce the appropriate glyph when the terminal is in the alternate character set mode. For example,

glyph name	vt100+ char	new tty char
upper left corner	l	R
lower left corner	m	F
upper right corner	k	T
lower right corner	j	G
horizontal line	q	,
vertical line	x	.

Now write down the characters left to right, as in "**acsc**=lRmFkTjGq\,x.".

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used. If the terminal does not have a pad character, specify **npc**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, **tparm(repeat_char, 'x', 10)** is the same as **xxxxxxxxxx**.

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some UNIX systems: If the environment variable **CC** exists, all occurrences of the prototype character are replaced with the character in **CC**.

Terminal descriptions that do not represent a specific kind of known terminal, such as **switch**, **dialup**, **patch**, and **network**, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to **virtual** terminal descriptions for which the escape sequences are known.) If the terminal is one of those supported by the UNIX system virtual terminal protocol, the terminal number can be given as **vt**. A line-turn-around sequence to be transmitted before doing reads should be specified in **rfi**.

If the terminal uses xon/xoff handshaking for flow control, give **xon**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted. Sequences to turn on and off xon/xoff handshaking may be given in **smxon** and **rmxon**. If the characters used for handshaking are not **^S** and **^Q**, they may be specified with **xonc** and **xoffc**.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm**#0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. A variation, **mc5p**, takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. If the text is not displayed on the terminal screen when the printer is on, specify **mc5i** (silent printer). All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Special Cases

The working model used by *terminfo* fits most terminals reasonably well. However, some terminals do not completely match that model, requiring special support by *terminfo*. These are not meant to be construed as deficiencies in the terminals; they are just differences between the working model and the actual hardware. They may be unusual devices or, for some reason, do not have all the features of the *terminfo* model implemented.

Terminals which can not display tilde (˜) characters, such as certain Hazeltine terminals, should indicate **hz**.

Terminals which ignore a linefeed immediately after an **am** wrap, such as the *Concept* 100, should indicate **xenl**. Those terminals whose cursor remains on the right-most column until another character has been received, rather than wrapping immediately upon receiving the right-most character, such as the VT100, should also indicate **xenl**.

If **el** is required to get rid of standout (instead of writing normal text on top of it), **xhp** should be given.

Those Teleray terminals whose tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This capability is also taken to mean that it is not possible to position the cursor on top of a "magic cookie" therefore, to erase standout mode, it is instead necessary to use delete and insert line.

Those Beehive Superbee terminals which do not transmit the escape or control−C characters, should specify **xsb**, indicating that the f1 key is to be used for escape and the f2 key for control−C.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be canceled by placing *xx*@ to the left of the capability definition, where *xx* is the capability. For example, the entry

```
att4424-2|Teletype 4424 in display function group ii,
     rev@, sgr@, smul@, use=att4424,
```

defines an AT&T 4424 terminal that does not have the
rev, **sgr**, and **smul** capabilities,
and hence cannot do highlighting.
This is useful for different modes for a terminal,
or for different user preferences.
More than one **use** capability may be given.

FILES

/usr/lib/terminfo/?/ *	compiled terminal description database
/usr/lib/.COREterm/?/ *	subset of compiled terminal description database
/usr/lib/tabset/ *	tab settings for some terminals, in a format appropriate to be output to the terminal (escape sequences that set margins and tabs)

SEE ALSO

captoinfo(1M), infocmp(1M), tic(1M), tput(1), curses(3X), printf(3S), term(5), tty(7).

Chapter 10 of the *Programmer's Guide*.

WARNING

As described in the "Tabs and Initialization" section above, a terminal's initialization strings, **is1**, **is2**, and **is3**, if defined, must be output before a *curses*(3X) program is run. An available mechanism for outputting such strings is **tput init** (see *tput*(1) and *profile*(4)).

Tampering with entries in **/usr/lib/.COREterm/?/ *** or **/usr/lib/terminfo/?/ *** (for example, changing or removing an entry) can affect programs such as *vi*(1) that expect the entry to be present and correct. In particular, removing the description for the "dumb" terminal will cause unexpected problems.

NOTE

The *termcap* database (from earlier releases of UNIX System V) may not be supplied in future releases.

This page is intentionally left blank

NAME

termtype.map – map from **NTC** TYPE name to terminology file

DESCRIPTION

termtype.map contains for each map entry the following informations:

termtype map name

filename of terminology table

This is an ASCII file using tabs or spaces as field separator. Each *termtype.map* entry is separated from the next by a new-line. The filename of the terminology table is the path and the name as placed in the directory **/etc/types** as required by the *terminology* program. The *termtype.map* file is only used by *getty* on lines connected to **NTC** (Network Terminal Controller).

This file resides in directory **/etc**.

FILES

/etc/termtype.map

SEE ALSO

getty(1), terminology(1).

This page is intentionally left blank

NAME

timezone – set default system time zone

SYNOPSIS

/etc/TIMEZONE

DESCRIPTION

This file sets and exports the time zone environmental variable **TZ**.

This file is "dotted" into other files that must know the time zone.

The syntax of **TZ** can be described as follows:

```
TZ            →          zone
                       | zone signed_time
                       | zone signed_time zone
                       | zone signed_time zone dst
zone          →          letter letter letter
signed_time   →          sign time
                       | time
time          →          hour
                       | hour : minute
                       | hour : minute : second
dst           →          signed_time
                       | signed_time ; dst_date , dst_date
                       | ; dst_date , dst_date
dst_date      →          julian
                       | julian / time
letter        →        a | A | b | B | ... | z | Z
hour          →        00 | 01 | ... | 23
minute        →        00 | 01 | ... | 59
second        →        00 | 01 | ... | 59
julian        →        001 | 002 | ...| 366
sign          →        − | +
```

EXAMPLES

The contents of **/etc/TIMEZONE** corresponding to the simple example below could be:

```
#       Time Zone
TZ=EST5EDT
export TZ
```

A simple setting for New Jersey could be

```
        TZ=EST5EDT
```

where **EST** is the abbreviation for the main time zone, **5** is the difference, in hours, between GMT (Greenwich Mean Time) and the main time zone, and **EDT** is the abbreviation for the alternate time zone.

The most complex representation of the same setting, for the year 1986, is:

```
TZ="EST5:00:00EDT4:00:00;117/2:00:00,299/2:00:00"
```

where **EST** is the abbreviation for the main time zone, **5:00:00** is the difference, in hours, minutes, and seconds between GMT and the main time zone, **EDT** is the abbreviation for the alternate time zone, **4:00:00** is the difference, in hours, minutes, and seconds between GMT and the alternate time zone, **117** is the number of the day of the year (Julian day) when the alternate time zone will take effect, **2:00:00** is the number of hours, minutes, and seconds past midnight when the alternate time zone will take effect, **299** is the number of the day of the year when the alternate time zone will end, and **2:00:00** is the number of hours, minutes, and seconds past midnight when the alternate time zone will end.

A southern hemisphere setting such as the Cook Islands could be

```
        TZ="KDT9:30KST10:00;64/5:00,303/20:00"
```

This setting means that **KDT** is the abbreviation for the main time zone, **KST** is the abbreviation for the alternate time zone, KST is **9** hours and **30** minutes later than GMT, KDT is **10** hours later than GMT, the starting date of KDT is the **64th** day at **5** AM, and the ending date of KDT is the **303rd** day at **8** PM.

Starting and ending times are relative to the alternate time zone. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be midnight.

Note that in most installations, **TZ** is set to the correct value by default when the user logs on, via the local */etc/profile* file (see *profile*(4)).

NOTES

When the longer format is used, the **TZ** variable must be surrounded by double quotes as shown.

The system administrator must change the Julian start and end days annually if the longer form of the **TZ** variable is used.

Setting the time during the interval of change from the main time zone to the alternate time zone or vice versa can produce unpredictable results.

SEE ALSO

rc2(1M), ctime(3C), profile(4), environ(5).

This page is intentionally left blank

NAME

unistd – file header for symbolic constants

SYNOPSIS

#include <unistd.h>

DESCRIPTION

The header file <*unistd.h*> lists the symbolic constants and structures not already defined or declared in some other header file.

```
/* Symbolic constants for the "access" routine: */

#define R_OK        4   /* Test for Read permission */
#define W_OK        2   /* Test for Write permission */
#define X_OK        1   /* Test for eXecute permission */
#define F_OK        0   /* Test for existence of File */

#define F_ULOCK     0   /* Unlock a previously locked region */
#define F_LOCK      1   /* Lock a region for exclusive use */
#define F_TLOCK     2   /* Test and lock a region for exclusive use */
#define F_TEST      3   /* Test a region for other processes locks */

/* Symbolic constants for the "lseek" routine: */

#define SEEK_SET 0   /* Set file pointer to "offset" */
#define SEEK_CUR 1   /* Set file pointer to current plus "offset" */
#define SEEK_END 2   /* Set file pointer to EOF plus "offset" */

/* Pathnames: */

#define GF_PATH    /etc/group    /* Pathname of the group file */
#define PF_PATH    /etc/passwd   /* Pathname of the passwd file */
```

This page is intentionally left blank

NAME

utmp, wtmp – utmp and wtmp entry formats

SYNOPSIS

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

DESCRIPTION

These files, which hold user and accounting information for such commands as *who*(1), *write*(1), and *login*(1), have the following structure as defined by **<utmp.h>**:

```
#define  UTMP_FILE      "/etc/utmp"
#define  WTMP_FILE      "/etc/wtmp"
#define  ut_name        ut_user

struct  utmp {
   char        ut_user[8];
               /* User login name */
   char        ut_id[4];
               /* /etc/inittab id (usually line #) */
   char        ut_line[12];
               /* device name (console, lnxx) */
   short       ut_pid;
               /* process id */
   short       ut_type;
               /* type of entry */
   struct      exit_status {
      short    e_termination;
               /* Process termination status */
      short    e_exit;
               /* Process exit status */
   } ut_exit;  /* The exit status of a process
                * marked as DEAD_PROCESS. */
   time_t      ut_time;
               /* time entry was made */
};
```

```
/*  Definitions for ut_type  */

#define EMPTY              0
#define RUN_LVL            1
#define BOOT_TIME          2
#define OLD_TIME           3
#define NEW_TIME           4
#define INIT_PROCESS       5
             /* Process spawned by "init" */
#define LOGIN_PROCESS      6
             /* A "getty" process waiting for login */
#define USER_PROCESS       7
             /* A user process */
#define DEAD_PROCESS       8
#define ACCOUNTING         9
#define UTMAXTYPE          ACCOUNTING
             /* Largest legal value of ut_type */
```

/* Special strings or formats used in the "ut_line" field */
/* when accounting for something other than a process. */
/* No string for the ut_line field can be more than 11 chars */
/* + a NULL in length */

```
#define RUNLVL_MSG     "run-level %c"
#define BOOT_MSG       "system boot"
#define OTIME_MSG      "old time"
#define NTIME_MSG      "new time"
```

FILES

/etc/utmp

/etc/wtmp

SEE ALSO

login(1), who(1), write(1), getut(3C).

NAME

intro – introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

This page is intentionally left blank

NAME

ascii – map of ASCII character set

DESCRIPTION

The following is a map of the ASCII character set.

The hexadecimal digits above the table give the high-order four bits of the character value. The hexadecimal digits to the left of the table give the low-order four bits of the character value.

	0	1	2	3	4	5	6	7
0	nul	dle	sp	0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
A	nl	sub	*	:	J	Z	j	z
B	vt	esc	+	;	K	[	k	{
C	ff	fs	,	<	L	\	l	|
D	cr	gs	-	=	M	]	m	}
E	so	rs	.	>	N	^	n	~
F	si	us	/	?	O	_	o	del

The characters are also found in the ISO 646 character set.

The following is a list of all the abbreviations used in the table:

ack	Acknowledge
bel	Bell
bs	Backspace
can	Cancel
cr	Carriage Return

dc1	Device Control 1 (X-ON)
dc2	Device Control 2
dc3	Device Control 3 (X-OFF)
dc4	Device Control 4
del	Delete
dle	Data Link Escape
em	End of Medium
enq	Enquiry
eot	End Of Transmission
esc	Escape
etb	End of Transmission Block
etx	End of Text
ff	Form Feed
fs	File Separator
gs	Group Separator
ht	Horizontal Tabulation
nak	Negative Acknowledge
nl	New Line (Line Feed)
nul	Null
rs	Record Separator
si	Shift-In
so	Shift-Out
soh	Start Of Heading
sp	Space
stx	Start of Text
sub	Substitute character
syn	Synchronous idle
us	Unit Separator
vt	Vertical Tabulation

SEE ALSO

iso-8859/1(5).

The *Supermax Virtual Terminal Guide.*

151

NAME

environ − user environment

DESCRIPTION

An array of strings called the "environment" is made available by *exec*(2) when a process begins. By convention, these strings have the form "name=value". The following names are used by various commands:

CFTIME The default format string to be used by the *date*(1) command and the **ascftime**( ) and **cftime**( ) routines (see *ctime*(3C)). If **CFTIME** is not set or is null, the default format string specified in the **/lib/cftime/***LANGUAGE* file (if it exists) is used in its place (see *cftime*(4)).

CHRCLASS A value that corresponds to a file in **/lib/chrclass** containing character classification and conversion information. This information is used by commands (such as *cat*(1), *ed*(1), *sort*(1), etc.) to classify characters as alphabetic, printable, upper case, etc. and to convert characters to upper or lower case.

When a program or command begins execution, the tables containing this information are initialized based on the value of **CHRCLASS**. If **CHRCLASS** is non-existent, null, set to a value for which no file exists in **/lib/chrclass**, or errors occur while reading the file, the ISO-8859/1 character set is used.

During execution, a program or command can change the values in these tables by calling the **setchrclass**() routine. For more detail, see *ctype*(3C).

These tables are created using the *chrtbl*(1M) command.

HOME	The name of the user's login directory, set by *login*(1) from the password file (see *passwd*(4)).
LANGUAGE	A language for which a printable file by that name exists in **/lib/cftime.** This information is used by commands (such as *date*(1), *ls*(1), *sort*(1), etc.) to print date and time information in the language specified. If **LANGUAGE** is non-existent, null, set to a value for which no file exists in **/lib/cftime**, or errors occur while reading the file, the last language requested will be used. (If no language has been requested, the language **usa_english** is assumed.) For a description of the content of files in **/lib/cftime**, see *cftime*(4).
PATH	The sequence of directory prefixes that *sh*(1), *time*(1), *nice*(1), *nohup*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (:). *login*(1) sets **PATH=:/bin:/usr/bin**. (For more detail, see the "Execution" section of the *sh*(1) manual page.)
TERM	The kind of terminal for which output is to be prepared. This information is used by commands, such as *mm*(1) or *vi*(1), which may exploit special capabilities of that terminal.
TZ	Time zone information. The simplest format is **xxx***n***zzz** where **xxx** is the standard local time zone abbreviation, *n* is the difference in hours from GMT (Greenwich Mean Time), and **zzz** is the abbreviation for an alternate time zone (usually the daylight-saving local time zone), if any; for example,

TZ="EST5EDT"

The most complex format allows you to specify the difference in hours of the alternate time zone from GMT and the starting day and time and ending day and time for using this alternate time zone. For example, in 1985 the complex format corresponding to the above simple example is:

TZ="EST5:00:00EDT4:00:00;118/2:00:00,300/2:00:00"

When the above complex format is used, it must be surrounded by double quotes. For more details, see *ctime*(3C) and *timezone*(4).

Further names may be placed in the environment by the *export* command and "name=value" arguments in *sh*(1), or by *exec*(2). It is unwise to conflict with certain shell variables that are frequently exported by **.profile** files: **MAIL**, **PS1**, **PS2**, **IFS** (see *profile*(4)).

NOTES

References to the *cftime*(4), *ctime*(3C), and *ctype*(3C) manual pages refer to programming capabilities available beginning with Issue 3.1 of the Software Development Utilities.

Administrators should note the following: if you attempt to set the current date to one of the dates that the standard and alternate time zones change (for example, the date that daylight time is starting or ending), and you attempt to set the time to a time in the interval between the end of standard time and the beginning of the alternate time (or the end of the alternate time and the beginning of standard time), the results are unpredictable.

SEE ALSO

cat(1), chrtbl(1M), date(1), ed(1), env(1), login(1), ls(1), nice(1), nohup(1), sh(1), sort(1), time(1), vi(1), exec(2), ctime(3C), ctype(3C), cftime(4), passwd(4), profile(4), timezone(4).

and

mm(1) in the *DOCUMENTER'S WORKBENCH Software Release 2.0 Technical Discussion and Reference Manual.*

NAME

fcntl – file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/* (The first three can only be set by open) */

#define O_RDONLY    0
#define O_WRONLY    1
#define O_RDWR      2
#define O_NDELAY    04     /* Non-blocking I/O */
#define O_APPEND    010
        /* append (writes guaranteed at the end) */
#define O_SYNC      020    /* synchronous write option */

/* Flag values accessible only to open(2) */

#define O_CREAT     00400
        /* open with file create (uses third open arg) */
#define O_TRUNC     01000 /* open with truncation */
#define O_EXCL      02000 /* exclusive open */

/* fcntl(2) requests */

#define F_DUPFD     0      /* Duplicate fildes */
#define F_GETFD     1      /* Get fildes flags */
#define F_SETFD     2      /* Set fildes flags */
#define F_GETFL     3      /* Get file flags */
#define F_SETFL     4      /* Set file flags */
#define F_GETLK     5      /* Get file lock */
#define F_SETLK     6      /* Set file lock */
#define F_SETLKW    7      /* Set file lock and wait */
#define F_CHKFL     8
        /* Check legality of file flag changes */
/* file segment locking control structure */
struct flock {
        short  l_type;
        short  l_whence;
        long   l_start;
        long   l_len;          /*if 0 then until EOF */
        short  l_sysid;        /*returned with F_GETLK*/
        short  l_pid;          /*returned with F_GETLK*/
}

/* file segment locking types */
#define  F_RDLCK   01      /*Read lock*/
#define  F_WRLCK   02      /*Write lock*/
#define  F_UNLCK   03      /*Remove locks*/
```

SEE ALSO

fcntl(2), open(2).

NAME

ISO 8859/1 – map of ISO 8859/1 character set

DESCRIPTION

The character set normally used on Supermax computers is the ISO 8859/1 international character set for Western Europe. The following is a map of that character set.

The hexadecimal digits above the table give the high-order four bits of the character value. The hexadecimal digits to the left of the table give the low-order four bits of the character value.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	nul	dle	sp	0	@	P	`	p	rfu	dcs	nbsp	°	À	Ð	à	ð
1	soh	dc1	!	1	A	Q	a	q	rfu	pu1	¡	±	Á	Ñ	á	ñ
2	stx	dc2	"	2	B	R	b	r	rfu	pu2	¢	²	Â	Ò	â	ò
3	etx	dc3	#	3	C	S	c	s	rfu	sts	£	³	Ã	Ó	ã	ó
4	eot	dc4	$	4	D	T	d	t	ind	cch	¤	´	Ä	Ô	ä	ô
5	enq	nak	%	5	E	U	e	u	nel	mw	¥	µ	Å	Õ	å	õ
6	ack	syn	&	6	F	V	f	v	ssa	spa	¦	¶	Æ	Ö	æ	ö
7	bel	etb	'	7	G	W	g	w	esa	epa	§	·	Ç	×	ç	÷
8	bs	can	(	8	H	X	h	x	hts	rfu	¨	¸	È	Ø	è	ø
9	ht	em	)	9	I	Y	i	y	htj	rfu	©	¹	É	Ù	é	ù
A	nl	sub	*	:	J	Z	j	z	vts	rfu	ª	º	Ê	Ú	ê	ú
B	vt	esc	+	;	K	[	k	{	pld	csi	«	»	Ë	Û	ë	û
C	ff	fs	,	<	L	\Å	l	|	plu	st	¬	¼	Ì	Ü	ì	ü
D	cr	gs	-	=	M	]	m	}	ri	osc	shy	½	Í	Ý	í	ý
E	so	rs	.	>	N	^	n	~	ss2	pm	®	¾	Î	Þ	î	þ
F	si	us	/	?	O	_	o	del	ss3	apc	¯	¿	Ï	ß	ï	ÿ

Actually, the table below was formed by merging several character sets: Characters with values 0x00-0x7f are found in the ASCII character set and in the ISO 646 character set.

Characters with values 0x20-0x7e are common to all ISO 8859 character sets. Characters with values 0x20-0x7e and 0xa0-0xff are found in the ISO 8859/1 character set. The control characters with values 0x80-0x9f are found in ANSI standard X3.64.

The following is a list of all the abbreviations used in the table:

ack	Acknowledge
apc	Application Program Command
bel	Bell
bs	Backspace
can	Cancel
cch	Cancel Character
cr	Carriage Return
csi	Control Sequence Introducer
dc1	Device Control 1 (X-ON)
dc2	Device Control 2
dc3	Device Control 3 (X-OFF)
dc4	Device Control 4
dcs	Device Control String
del	Delete
dle	Data Link Escape
em	End of Medium
enq	Enquiry
eot	End Of Transmission
epa	End of Protected Area
esa	End of Selected Area
esc	Escape
etb	End of Transmission Block
etx	End of Text
ff	Form Feed
fs	File Separator
gs	Group Separator
ht	Horizontal Tabulation
htj	Horizontal Tabulation with Justification
hts	Horizontal Tabulation Set
ind	Index

158

mw	Message Waiting
nak	Negative Acknowledge
nbsp	No-Break Space
nel	Next Line
nl	New Line (Line Feed)
nul	Null
osc	Operating System Command
pld	Partial Line Down
plu	Partial Line Up
pm	Privacy Message
pu1	Private Use 1
pu2	Private Use 2
rfu	Reserved for Future Use
ri	Reverse Index
rs	Record Separator
shy	Soft Hyphen
si	Shift-In
so	Shift-Out
soh	Start Of Heading
sp	Space
spa	Start of Protected Area
ss2	Single Shift 2
ss3	Single Shift 3
ssa	Start of Selected Area
st	String Terminator
sts	Set Transmit State
stx	Start of Text
sub	Substitute character
syn	Synchronous idle
us	Unit Separator
vt	Vertical Tabulation
vts	Vertical Tabulation Set

SEE ALSO

ascii(5).

The *Supermax Virtual Terminal Guide.*

This page is intentionally left blank

NAME

math – math functions and constants

SYNOPSIS

#include <math.h>

DESCRIPTION

This file contains declarations of all the functions in the Math Library (described in Section 2&3), as well as various functions in the C Library (Section 2&3) that return floating-point values.

It defines the structure and constants used by the *matherr*(3M) error-handling mechanisms, including the following constant used as an error-return value:

HUGE	The maximum value of a single-precision floating-point number.

The following mathematical constants are defined for user convenience:

M_E	The base of natural logarithms (*e*).
M_LOG2E	The base-2 logarithm of *e*.
M_LOG10E	The base-10 logarithm of *e*.
M_LN2	The natural logarithm of 2.
M_LN10	The natural logarithm of 10.
M_PI	π, the ratio of the circumference of a circle to its diameter.
M_PI_2	$\frac{\pi}{2}$
M_PI_4	$\frac{\pi}{4}$
M_1_PI	$\frac{1}{\pi}$
M_2_PI	$\frac{2}{\pi}$

M_2_SQRTPI	$\frac{2}{\sqrt{\pi}}$
M_SQRT2	The positive square root of 2.
M_SQRT1_2	The positive square root of 1/2.

For the definitions of various machine-dependent "constants," see the description of the <*values.h*> header file.

SEE ALSO

intro(2&3), matherr(3M), values(5).

NAME

prof – profile within a function

SYNOPSIS

#define MARK
#include <prof.h>

void MARK (name)

DESCRIPTION

MARK will introduce a mark called *name* that will be treated the same as a function entry point. Execution of the mark will add to a counter for that mark, and program-counter time spent will be accounted to the immediately preceding mark or to the function if there are no preceding marks within the active function.

Name may be any combination of numbers or underscores. Each *name* in a single compilation must be unique, but may be the same as any ordinary program symbol.

For marks to be effective, the symbol MARK must be defined before the header file <*prof.h*> is included. This may be defined by a preprocessor directive as in the synopsis, or by a command line argument, i.e:

```
cc -p -DMARK foo.c
```

If MARK is not defined, the *MARK*(name) statements may be left in the source files containing them and will be ignored.

EXAMPLE

In this example, marks can be used to determine how much time is spent in each loop. Unless this example is compiled with *MARK* defined on the command line, the marks are ignored.

```
#include <prof.h>
foo( )
{
        int i, j;
        .
        .
```

```
                .
                .
                MARK(loop1);
                for (i = 0; i          < 2000; i++) {
                . . .
                }
                MARK(loop2);
                for (j = 0; j          < 2000; j++) {
                . . .
                }
        }
```

SEE ALSO

prof(1), profil(2), monitor(3C).

NAME

regexp − regular expression compile and match routines

SYNOPSIS

#define INIT <declarations>
#define GETC( ) <getc code>
#define PEEKC( ) <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile (instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
int eof;

int step (string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;

DESCRIPTION

This page describes general-purpose regular expression matching routines in the form of *ed*(1), defined in **<regexp.h>** . Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), *expr*(1), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC( )	Return the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.

PEEKC( )	Return the next character in the regular expression. Successive calls to PEEKC( ) should return the same character [which should also be the next character returned by GETC( )].
UNGETC(*c*)	Cause the argument *c* to be returned by the next call to GETC( ) [and PEEKC( )]. No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC( ). The value of the macro UNGETC(*c*) is always ignored.
RETURN(*pointer*)	This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.
ERROR(*val*)	This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf*−*expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each program that includes this file must have a **#define** statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for GETC( ), PEEKC( ) and UNGETC( ). Otherwise it can be used to declare external variables that might be used by GETC( ), PEEKC( ) and UNGETC( ). See the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

```
step(string, expbuf)
```

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, *loc1* will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* will try to match the regular expression to the beginning of the string only. If more than one regular expression is to be compiled before the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns non-zero indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called; simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, *advance* will break out of the loop that backs up and will return zero. This is used by *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like **s/y*//g** do not loop forever.

The additional external variables *sed* and *nbra* are used for special purposes.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT        register char *sp = instring;
#define GETC()      (*sp++)
#define PEEKC()     (*sp)
#define UNGETC(c)   (--sp)
#define RETURN(c)   return;
#define ERROR(c)    regerr()
```

```
#include  <regexp.h>
...
          (void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
          if (step(linebuf, expbuf))
                              succeed( );
```

SEE ALSO

ed(1), expr(1), grep(1), sed(1).

NAME

stat – data returned by stat system call

SYNOPSIS

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

DESCRIPTION

The system calls *stat* and *fstat* return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

Structure of the result of stat

```
struct  stat
{
        dev_t    st_dev;
        ushort   st_ino;
        ushort   st_mode;
        short    st_nlink;
        ushort   st_uid;
        ushort   st_gid;
        dev_t    st_rdev;
        off_t    st_size;
        time_t   st_atime;
        time_t   st_mtime;
        time_t   st_ctime;
};

#define  S_IFMT   0170000 /* type of file */
#define  S_IFDIR  0040000 /* directory */
#define  S_IFCHR  0020000 /* character special */
#define  S_IFBLK  0060000 /* block special */
#define  S_IFREG  0100000 /* regular */
#define  S_IFIFO  0010000 /* fifo */
#define  S_ISUID  04000
                   /* set user id on execution */
#define  S_ISGID  02000
                   /* set group id on execution */
```

```
#define   S_IREAD   00400
              /* read permission, owner */
#define   S_IWRITE  00200
              /* write permission, owner */
#define   S_IEXEC   00100
              /* execute/search permission, owner */
#define   S_ENFMT   S_ISGID
              /* record locking enforcement flag */
#define   S_IRWXU   00700
              /* read,write, execute: owner */
#define   S_IRUSR   00400
              /* read permission: owner */
#define   S_IWUSR   00200
              /* write permission: owner */
#define   S_IXUSR   00100
              /* execute permission: owner */
#define   S_IRWXG   00070
              /* read, write, execute: group */
#define   S_IRGRP   00040
              /* read permission: group */
#define   S_IWGRP   00020
              /* write permission: group */
#define   S_IXGRP   00010
              /* execute permission: group */
#define   S_IRWXO   00007
              /* read, write, execute: other */
#define   S_IROTH   00004
              /* read permission: other */
#define   S_IWOTH   00002
              /* write permission: other */
#define   S_IXOTH   00001
              /* execute permission: other */
```

SEE ALSO

stat(2), types(5).

NAME

term − conventional names for terminals

DESCRIPTION

These names are used by certain commands (e.g., *man*(1), *tabs*(1), *tput*(1), *vi*(1) and *curses*(3X)) and are maintained as part of the shell environment in the environment variable **TERM** (see *sh*(1), *profile*(4), and *environ*(5)).

Entries in *terminfo*(4) source files consist of a number of comma-separated fields. (To obtain the source description for a terminal, use the **−I** option of *infocmp*(1M).) White space after each comma is ignored. The first line of each terminal description in the *terminfo*(4) database gives the names by which *terminfo*(4) knows the terminal, separated by bar (|) characters. The first name given is the most common abbreviation for the terminal (this is the one to use to set the environment variable **TERMINFO** in *$HOME/.profile*; see *profile*(4)), the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should contain no blanks and must be unique in the first 14 characters; the last name may contain blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, for example, for the AT&T 4425 terminal, **att4425**. This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Up to 8 characters, chosen from [a−z0−9], make up a basic terminal name. Names should generally be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name. Terminal sub-models, operational modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. Thus, an AT&T 4425 terminal in 132 column mode would be **att4425−w**. The following suffixes should be used where possible:

Suffix	Meaning	Example
−w	Wide mode (more than 80 columns)	att4425−w
−am	With auto. margins (usually default)	vt100−am
−nam	Without automatic margins	vt100−nam
−n	Number of lines on the screen	aaa−60
−na	No arrow keys (leave them in local)	c100−na
−n**p**	Number of pages of memory	c100−4p
−rv	Reverse video	att4415−rv

To avoid conflicts with the naming conventions used in describing the different modes of a terminal (e.g., **−w**), it is recommended that a terminal's root name not contain hyphens. Further, it is good practice to make all terminal names used in the *terminfo*(4) database unique. Terminal entries that are present only for inclusion in other entries via the **use=** facilities should have a '+' in their name, as in **4415+nl**.

Some of the known terminal names may include the following generic terminfo table supported by the VTI concept (for a complete list, type: **ls -C /usr/lib/terminfo/?**):

T*	Generic terminfo table supported by the VTI concept.
2621,hp2621	Hewlett-Packard 2621 series
2631	Hewlett-Packard 2631 line printer
2631−c	Hewlett-Packard 2631 line printer − compressed mode
2631−e	Hewlett-Packard 2631 line printer − expanded mode
2640,hp2640	Hewlett-Packard 2640 series
2645,hp2645	Hewlett-Packard 2645 series
3270	IBM Model 3270
33,tty33	AT&T Teletype Model 33 KSR
35,tty35	AT&T Teletype Model 35 KSR
37,tty37	AT&T Teletype Model 37 KSR
4000a	Trendata 4000a

4014,tek4014	TEKTRONIX 4014
40,tty40	AT&T Teletype Dataspeed 40/2
43,tty43	AT&T Teletype Model 43 KSR
4410,5410	AT&T 4410/5410 terminal in 80-column mode – version 2
4410–nfk,5410–nfk	AT&T 4410/5410 without function keys – version 1
4410–nsl,5410–nsl	AT&T 4410/5410 without pln defined
4410–w,5410–w	AT&T 4410/5410 in 132-column mode
4410v1,5410v1	AT&T 4410/5410 terminal in 80-column mode – version 1
4410v1–w,5410v1–w	AT&T 4410/5410 terminal in 132-column mode – version 1
4415,5420	AT&T 4415/5420 in 80-column mode
4415–nl,5420–nl	AT&T 4415/5420 without changing labels
4415–rv,5420–rv	AT&T 4415/5420 80 columns in reverse video
4415–rv–nl,5420–rv–nl	AT&T 4415/5420 reverse video without changing labels
4415–w,5420–w	AT&T 4415/5420 in 132-column mode
4415–w–nl,5420–w–nl	AT&T 4415/5420 in 132-column mode without changing labels
4415–w–rv,5420–w–rv	AT&T 4415/5420 132 columns in reverse video
4415–w–rv–nl, 5420–w–rv–nl	AT&T 4415/5420 132 columns reverse video without changing labels
4418,5418	AT&T 5418 in 80-column mode
4418–w,5418–w	AT&T 5418 in 132-column mode
4420	AT&T Teletype Model 4420
4424	AT&T Teletype Model 4424
4424-2	AT&T Teletype Model 4424 in display function group ii
4425,5425	AT&T 4425/5425
4425–fk,5425–fk	AT&T 4425/5425 without function

	keys
4425−nl,5425−nl	AT&T 4425/5425 without changing labels in 80-column mode
4425−w,5425−w	AT&T 4425/5425 in 132-column mode
4425−w−fk,5425−w−fk	AT&T 4425/5425 without function keys in 132-column mode
4425−nl−w,5425−nl−w	AT&T 4425/5425 without changing labels in 132-column mode
4426	AT&T Teletype Model 4426S
450	DASI 450 (same as Diablo 1620)
450−12	DASI 450 in 12-pitch mode
500,att500	AT&T-IS 500 terminal
510,510a	AT&T 510/510a in 80-column mode
513bct,att513	AT&T 513 bct terminal
5320	AT&T 5320 hardcopy terminal
5420_2	AT&T 5420 model 2 in 80-column mode
5420_2−w	AT&T 5420 model 2 in 132-column mode
5620,dmd	AT&T 5620 terminal 88 columns
5620−24,dmd−24	AT&T Teletype Model DMD 5620 in a 24x80 layer
5620−34,dmd−34	AT&T Teletype Model DMD 5620 in a 34x80 layer
610,610bct	AT&T 610 bct terminal in 80-column mode
610−w,610bct−w	AT&T 610 bct terminal in 132-column mode
7300,pc7300,unix_pc	AT&T UNIX PC Model 7300
735,ti	Texas Instruments TI735 and TI725
745	Texas Instruments TI745
dumb	generic name for terminals that lack reverse line-feed and other special escape sequences
hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer

pt505	AT&T Personal Terminal 505 (22 lines)
pt505 – 24	AT&T Personal Terminal 505 (24-line mode)
sync	generic name for synchronous Teletype Model 4540-compatible terminals.

Commands whose behavior depends on the type of terminal should accept arguments of the form **–T***term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable **TERM**, which, in turn, should contain *term*.

FILES

/usr/lib/terminfo/?/ * compiled terminal description database

SEE ALSO

infocmp(1M), man(1), sh(1), stty(1), tabs(1), tplot(1G), tput(1), vi(1), curses(3X), profile(4), terminfo(4), environ(5).

Chapter 10 of the *Programmer's Guide.*

NOTES

Not all programs follow the above naming conventions.

This page is intentionally left blank

NAME

types – primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in UNIX system code; some data are accessible to user code:

```
typedef  struct { int r[1]; } *physadr;
typedef  long                 daddr_t;
typedef  char *               caddr_t;
typedef unsigned char         unchar;
typedef unsigned short        ushort;
typedef  unsigned int         uint;
typedef  unsigned long        ulong;
typedef  ushort               ino_t;
typedef  short                cnt_t;
typedef  long                 time_t;
typedef  int                  label_t[13];
typedef  short                dev_t;
typedef  long                 off_t;
typedef  long                 paddr_t;
typedef int                   key_t;
typedef unsigned char         use_t;
typedef short                 sysid_t;
typedef short                 index_t;
typedef short                 lock_t;
typedef unsigned int          size_t;
```

The form *daddr_t* is used for disk addresses except in an i-node on disk, see *fs*(4). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

SEE ALSO

fs(4).

This page is intentionally left blank

NAME

values – machine-dependent values

SYNOPSIS

#include <values.h>

DESCRIPTION

This file contains a set of manifest constants, conditionally defined for particular processor architectures.

The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit.

BITS(*type*)	The number of bits in a specified type (e.g., int).
HIBITS	The value of a short integer with only the high-order bit set (0x8000).
HIBITL	The value of a long integer with only the high-order bit set (0x80000000).
HIBITI	The value of a regular integer with only the high-order bit set (the same as HIBITL).
MAXSHORT	The maximum value of a signed short integer (0x7FFF ≡ 32767).
MAXLONG	The maximum value of a signed long integer (0x7FFFFFFF ≡ 2147483647).
MAXINT	The maximum value of a signed regular integer (the same as MAXLONG).
MAXFLOAT, LN_MAXFLOAT	The maximum value of a single-precision floating-point number, and its natural logarithm.
MAXDOUBLE, LN_MAXDOUBLE	The maximum value of a double-precision floating-point number, and its natural logarithm.

MINFLOAT, LN_MINFLOAT	The minimum positive value of a single-precision floating-point number, and its natural logarithm.
MINDOUBLE, LN_MINDOUBLE	The minimum positive value of a double-precision floating-point number, and its natural logarithm.
FSIGNIF	The number of significant bits in the mantissa of a single-precision floating-point number.
DSIGNIF	The number of significant bits in the mantissa of a double-precision floating-point number.

SEE ALSO

intro(3), math(5).

NAME

varargs – handle variable argument list

SYNOPSIS

#include <varargs.h>

va_alist

va_dcl

void va_start(pvar)
va_list pvar;

type **va_arg(pvar,** *type***)**
va_list pvar;

void va_end(pvar)
va_list pvar;

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists [such as *printf*(3S)] but do not use *varargs* are inherently nonportable, as different machines use different argument-passing conventions.

va_alist is used as the parameter list in a function header.

va_dcl is a declaration for *va_alist*. No semicolon should follow *va_dcl*.

va_list is a type defined for the variable used to traverse the list.

va_start is called to initialize *pvar* to the beginning of the list.

va_arg will return the next argument in the list pointed to by *pvar*. *Type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

va_end is used to clean up.

Multiple traversals, each bracketed by *va_start* ... *va_end*, are possible.

EXAMPLE

This example is a possible implementation of *execl*(2).

```
#include <varargs.h>
#define MAXARGS100

/* execl is called by
        execl(file, arg1, arg2, ..., (char *)0);
*/
execl(va_alist)
va_dcl
{
    va_list ap;
    char *file;
    char *args[MAXARGS];
    int argno = 0;

    va_start(ap);
    file = va_arg(ap, char *);
    while ((args[argno++] = va_arg(ap, char *)) != (char *)0)
          ;
    va_end(ap);
    return execv(file, args);
}
```

SEE ALSO

exec(2), printf(3S), vprintf(3S).

NOTES

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this from the stack frame.

For example, *execl* is passed a zero pointer to signal the end of the list. *Printf* can tell how many arguments are there by the format.

It is non-portable to specify a second argument of *char*, *short*, or *float* to *va_arg*, since arguments seen by the called function are not *char*, *short*, or *float*. C converts *char* and *short* arguments to *int* and converts *float* arguments to *double* before passing them to a function.

This page is intentionally left blank

NAME

intro – Introduction to games.

DESCRIPTION

This section describes the recreational and educational programs found in the directory **/usr/games**.

The use of these programs may be rejected during business hours by using the **cron**(1M) if so decided by the system administrator.

BUGS

Many of these programs contains bugs, but little or no attempt will be made to correct these bugs.

This page is intentionally left blank

NAME

aliens – the alien invaders attack the earth.

SYNOPSIS

usr/games/aliens
usr/games/alienslog

DESCRIPTION

aliens is a Supermax version of Space Invaders. The program is almost self-explanatory.

alienslog displays the high-score list.

FILES

/usr/games/lib/alienslog	Score file.

BUGS

The program may not run correctly on terminals running slower than 9600 baud.

This page is intentionally left blank

NAME

arithmetic – provides drill in number facts

SYNOPSIS

/usr/games/arithmetic [±] [range]

DESCRIPTION

arithmetic types out simple arithmetic problems and waits for an answer to be typed in. If the answer is correct you will receive the word **Right** and be given a new problem. If the answer is wrong you will receive the word **What**, and the program will wait for another answer. After every twentieth question statistics on correctness and respond time used will be shown on the screen.

Type an interrupt (CTRL/C) to quit the program.

The first optional determines the kind of program to be generated. +, –, **x** and /, respectively will cause addition, subtraction, multiplication and division questions to be generated. One or more characters can be given. If more than one character is given, the different types of questions will be mixed in random order. Default is ±

range is a decimal number. All addends, subtrahends, differences, multiplicands, divisors and quotients will be less than or equal to the value of *range*. Default *range* is 10.

At start all numbers less than or equal to *range* are equally likely to appear. If the respondent makes a mistake, the numbers in a missed question becomes more likely to reappear.

As a matter of educational philosophy the program will not submit correct answers, since the trainee in principle should be able to figure out the correct answers. The program is intended to provide drill for someone beyond the first schooldays and not to teach number fact *de novo*. For almost all users the relevant statistics should be consumed time per question, and not the correct percentage.

This page is intentionally left blank

NAME

back – The game of backgammon

SYNOPSIS

/usr/games/back

DESCRIPTION

back is a program which provides a partner for the game of backgammon. It is designed to be played at three different levels of skill, one of which you must select. In addition to selecting the opponent's level you may also indicate, that you want to throw your own dice during your turns, (for the superstitious players!). You will also be given the opportunity to make the first move. The practice if each player throwing one dice for the first move is not incorporated.

The points are numbered 1–24 with 1 being 'white's extreme inner table; 24 being 'brown's inner table; 0 (zero) being the bar for removed 'white' pieces, and 25 the bar for 'brown'.

For details on how moves are expressed, type **y** when *back* at the beginning of the game asks if you want `Instructions?`.

When *back* first asks `Move?` you may type **?** to get a list of the possible move options other than entering your numerical move.

BUGS

You should always use the 'move's in the order indicated by the display of the dice. If your dice shows the figures of 4 and 2, use the number 4 first.

This page is intentionally left blank

NAME

bj – The game of Black Jack

SYNOPSIS

/usr/games/bj

DESCRIPTION

bj is a serious attempt of simulating the dealer in the game of Black Jack (or Twenty–One) as might be found in Reno in Las Vegas. The following rules apply:

The bet is $2 on every hand:

A player ”natural” (Black Jack) pays $3. A dealer ”natural” looses $2. Both dealer and player naturals are a ”push” (no money exchange).

If the dealer has an ace up the player is allowed to make an ”insurance” bet against the chance of a dealer natural. If the bet is not taken the play will resume as normal. If the bet is taken, it is considered as a ’side bet’, where the player wins $2 if the dealer has a natural, or, if the dealer does not have any natural, the players loss will be $1.

If the player receives two cards showing the same value, he is allowed to ”double”. This means the player will be allowed to play two hands, each with one of these cards. (The bet is also doubled with $2 on each hand).

If a dealt hand has a total of 10 or 11 the player is allowed to ”double down”. The bet may be doubled from $2 to $4, and the player will receive exactly one more card on that hand.

During normal play the player may ”hit” (draw a card), as long as the total do not go beyond ’twenty–one’. If the player ”busts” (exceeds ’twenty–one’), then the dealer wins the bet.

When the player ”stands” (decides not to ”hit”), the dealer hits until he attains a total of 17 or more. If the dealer ”busts” the player wins the bet.

If both player and dealer stands, then the one with the highest total winds. A tie is a push.

The machine deals and keeps score. The following questions will be asked at appropriate times. Each question is answered by either a **y** followed by <Return> for 'yes', or just by <Return> indicating 'no'.

```
?      (meaning:"Do you want a hit?")
Insurance?
Double down?
```

Every time the deck is shuffled, the dealer states and the "action" (total bet) and "standing" (total won or lost) is printed.

To quit the program and exit, type an interrupt (CTRL/C) and the action and standing will be printed.

NAME

craps – The game of craps

SYNOPSIS

/usr/games/craps

DESCRIPTION

craps is a form of the Game of Craps as played in Las Vegas. The program simulates the *roller*, while the user (the player) places the bets. The player may at any time choose to bet with the *roller*, or the *House*. A bet of a negative amount is taken as the bet with the *House*. Any other bet is a bet with the *roller*.

The player begins with a "bankroll" of $2,000.

The program prompts with:

```
bet?
```

The bet can be all or part of the player's bankroll. Any bet over the total bankroll is rejected and the program prompts with "`bet?`" until a proper bet is made.

Once the bet is accepted the *roller* throws the dice. The following rules apply, (the player wins or loses depending on whether the bet is placed with the *roller* or with the *House*. The odds are even). The *first* roll is the roll immediately following a bet.

1. On the first roll:

7 or 11	wins for the roller,
2, 3 or 12	wins for the House,
any other number	is the *point*, roll again (Rule 2 applies).

2. On subsequent rolls:

point	roller wins,
7	House wins,
any other number	roll again.

If a player loses the entire bankroll, the *House* will offer to lend the player an additional $2,000. The program will prompt:

```
marker?
```

A "yes" (or "y") consummates the loan. Any other reply terminates the game.

If a player owes the *House* money, the *House* will before a bet is placed, remind the player how many markers are outstanding.

If at any time the bankroll of a player with outstanding markers, exceeds $2,000 the *House* asks:

```
Repay marker?
```

A reply of "yes" (or "y") indicates the player's willingness to repay the loan. If only one marker is outstanding, it is immediately repaid. If, however, more than one marker are outstanding, the *House* asks:

```
How many?
```

markers the player would like to repay. If an invalid number is entered (or just a <RETURN>), an appropriate message is printed and the program will prompt with "*How many*?" until a valid number is entered.

If a player accumulates 10 markers, (a total of $20,000 borrowed from the *House*), the program informs the player of the situation and exits.

Should the bankroll of a player, who has outstanding markers, exceed $50,000, the *total* amount of money borrowed will *automatically* be repaid to the *House*.

Any player who accumulates $100,000 or more breaks the bank. The program then prompts:

```
New game?
```

to give the *House* a chance to win its money back.

Any reply other than ”yes” is considered as ”no”, (except in the case of ”bet?” or ”How many?”).

To exit the program use an interrupt (CTRL/C) or CTRL/D. The program will indicate whether the player won, lost or broke even.

MISCELLANEOUS

The random number generator for the dice numbers uses the seconds from the time of the day. Depending of system usage these numbers may sometimes seem strange, but occurences of this type in a real dice situation are not uncommon.

This page is intentionally left blank

NAME

fish – play the game of "Go Fish"

SYNOPSIS

/usr/games/fish

DESCRIPTION

fish plays the game of "Go Fish", a children's card game. The object is to accumulate 'books' of 4 cards with the same face value. The players alternate turns. Each turn begins with one player selecting a card from his hand, and asking the other player for all cards of that face value. If the other player has one or more cards of that face value on his hand, he gives them to the first player and the first player makes another request.

Eventually, the first player asks for a card which is not in the second player's hand. The second player then replies *"Go Fish"*! The first player then draws a card from the 'pool' of undealt vards. If this is the card requested by the first player, he draws again.

When a book is made, either through drawing or through requesting, the cards are laid down and no further action takes place with that face value.

To play the computer simply make guesses by typing: *a, 2, 3, 4, 5, 6, 7, 8, 9, 10, j, q,* or *k*, when asked. By hitting <Return> you may obtain information about the size of the hand; the pool and about achieved books.

Typing a "p" as the first guess puts you into a 'pro' level; the default is pretty dumb.

This page is intentionally left blank

NAME

hack – exploring The Dungeons of Doom

SYNOPSIS

/usr/games/hack [–TSKFCW] [–u *playername*]

/usr/games/hack –s [–TSKFCW]

/usr/games/hack –s [*playernames*]

DESCRIPTION

hack is a display oriented Dungeons and Dragons-like game.

To get started you really need two commands. The command **?** will give you a list of all available commands, and the command **/** will identify what you see on the screen.

To win the game, (as opposed to merely playing to bet other peoples high scores), you must locate the *Amulet of Yendor*, which is somewhere below the 20th level of the Dungeon and get it out. Nobody has achieved this yet, and if somebody does he will probably go down in history as a hero among heros.

When the game ends, either by your 'death'; if you quit, or if you escape from the caves, *hack* will show you (a fragment) of the list of top scores.

The scoring is based on many aspects of your behaviour, but a rough estimate is obtained by taking the amount of gold you have found in the cave, plus four times your (real) experience.

Precious stones may be worth a lot of gold when brought to the exit. There is a 10% penalty fee for getting yourself killed.

The **–u** *playername* option supplies the answer to the question "*Who are you*?

The options **–T, –S, –K, –F, –C, –W** may be used to supply the answer to the question "*What kind of character* ...?"

The **–s** option will print out a list of your scores. It may be followed by an option **–T, –S, –K, –F, –C, –W** to print the scores of Tourists, Speleologists, Knights, Fighters, Cavemen, and **Wizards**, respectively. It may alternatively be followed by one or more players names to print the scores of the players mentioned.

AUTHORS

Jay Fenlason (+ Kenny Woodland, Mike Thome and John Payne), wrote the original 'hack', very much like rogue, (but not full of bugs).

Andries Brouwer continously deformed their sources into the current version – in fact an entirely different game.

FILES

In the directory **/usr/games/lib/hackdir** you will find all files used by *hack* located. The file **/usr/games/hack** itself is a shell script; it can be edited to move the files used by *hack*. The files in **/usr/games/lib/hackdir** includes:

hack	The hack program.
data, rumors	Data files used by hack.
help, hh	Help data files.
record	The list of top scores.
save	A subdirectory containing the saved games.
bones_dd	Descriptions of the ghost and belongings of a deceased adventurer.
xlock.dd	Description of a Dungeon level.
safelock	Lock file for xlock.
record_lock	Lock file for record.

ENVIRONMENT

USER or **LOGNAME**	Your login name.
HOME	Your home directory.
SHELL	Your shell.
TERM	The type of your terminal.

HACKPAGER, PAGER	Pager used instead of default pager.
MAIL	Mailbox file.
MAILREADER	Reader used instead of default (probably /bin/mail or /usr/ucb/mail).
HACKDIR	Playground.
HACKOPTIONS	String predefining several hack options (see help file).

BUGS

Probably infinite.

This page is intentionally left blank

NAME

life – play the Game of Life

SYNOPSIS

/usr/games/life [–r]

DESCRIPTION

life is a pattern generating game set up for interactive use on a video terminal. The way it operates is: You use a series of commands to set up a pattern on the screen and let it generate further patterns from that pattern.

The algorithm used is: For each square in the matrix, look at it and its eight adjacent neighbours. If the present square is not occupied and exactly three of its neighbour squares are occupied, then that square will be occupied in the next pattern. If the present is occupied and two or three of its neighbour squares are occupied, then that square will be occupied in its next pattern.

The edges of the screen are normally treated as an unoccupied void. If you specify the **–r** option on the command line, the screen is treated as a sphere; that is, the top and bottom lines are considered adjacent and the left and right columns are considered adjacent.

The pattern generation number and the number of occupied squares are displayed in the lower left corner.

Below is a list of commands available to the user. An # stands for any number. An ^ followed by a capital letter represents a control character.

#,#a	Add a block of elements. The first number specifies the horizontal width. The second number specifies the vertical width. If a number is not specified, the default is 1.
#c	Step through the # patterns. If no number is specified, step forever. The operation can be aborted by typing 'rubout' (delete).

#,#d	Delete a block of elements. The first number specifies the horizontal width. The second number specifies the vertical width. If a number is not specified, the default is 1.
#f	Generate a small flyer at the present location. The number (modulo 8) determines the direction.
#,#g	Move to absolite screen location. The first number specifies the horizontal location. The second number specifies the vertical location. If a number is not specified, the default is 0 (zero).
#h	Move left # steps. If no number is specified, the default is 1.
#j	Move down # steps. The default is 1.
#k	Move up # steps. The default is 1.
#l	Move right # steps. The default is 1.
#n	Step through the next # patterns. If no number is specified, generate the next pattern. The operation can be aborted by typing 'rubout' (delete).
p	Put the last yanked or deleted block at the present location.
q	Quit.
#,#y	Yank a block of elements. The first number specifies the horizontal width. The second number specifies the vertical width. If a number is not specified, the default is 1.
C	Clear the pattern.
#F	Generate a big flyer at the present location. The number (modulo 8) determines the location.
#H	Move to the left margin.
#J	Move to the bottom margin.
#K	Move to the top margin.

#L	Move to the right margin.
#^H	Move left **#** steps. If no number is specified, the default is 1.
#^J	Move down **#** steps. The default is 1.
#^K	Move up **#** steps. The default is 1.
#^L	Move right **#**steps. The default is 1.
^R	Redraw the screen. This is used for occations where the terminal screws up.
.	Repeat the last add (a) or delete (d) operation.
;	Repeat the last move, (**h, j, k, l**) operation.

AUTHOR

Asa Romberger.

BUGS

The following features are planned, but not implemented:

#,#S	Save the selected area in a file.
R	Restore from a file.
m	Generate a macro command.
!	Shell escape.
e	Edit a file.
i	Input commands from a file.

This page is intentionally left blank

NAME

robots – escape from the robots

SYNOPSIS

/usr/games/robots

DESCRIPTION

The object of the game *robots* is to move around inside a box on the screen without getting eaten by the robots chasing you, and without bumping or running into anything.

If a robot runs into another robot while chasing you, they crash and leave a junk heap.

You start out with 10 robots each worth 10 points. If you defeat all of them, you get 20 robots each worth 20 points. The 30 robots each worth 30 points, etc., until you get eaten!!

The game keeps track of the top ten scores and prints these top ten scores at the end of the game.

The valid commands are described on the screen.

This page is intentionally left blank

NAME

ttt – the game of Tic–Tac–Toe

SYNOPSIS

/usr/games/ttt [–ie]

DESCRIPTION

The *ttt* is the ○ and X game, really popular in the first grade. The *ttt* is an educational program never making the same mistake twice.

Although it learns – it learns slowly. The program must loose nearly 80 games to know the entire game completely. Specifying the **–e** option will cause the program to learn somewhat faster.

The **–i** option will cause the *ttt* to print out instructions.

Terminate the program by pressing the interrupt key (CTRL/C).

ttt reads the file **/usr/games/ttt.a** for knowledge about the game. It creates in the current directory a new file **ttt.a** containing its previous knowledge plus some added knowledge. Copying **ttt.a** into the **/usr/games/ttt.a** makes *ttt* smarter when played the next time.

FILES

/usr/games/ttt.a, ttt.a

This page is intentionally left blank

NAME

intro – introduction to special files

DESCRIPTION

This section describes various special files that refer to specific hardware peripherals, and operating system device drivers. STREAMS [see *intro*(2)] software drivers, modules and the STREAMS-generic set of *ioctl*(2) system calls are also described.

For hardware related files, 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 operating system device driver are discussed where applicable.

Where file names are given as, for example, **/dev/dsk/u#c#s#**, the number signs (#) should be replaced by decimal numbers.

Many entries specify the device number of each device. The device number is a 2-byte entity whose value is derived from the numbers that replace the number signs in the device names. The most significant byte of the device number is called the major device number. The least significant byte is called the minor device number.

This page is intentionally left blank

NAME

cioc − the CIOC devices

SYNOPSIS

/dev/cioc/u#c#s#
/dev/cioctl/u#

DESCRIPTION

A device with the name **/dev/cioc/u***N***c***C***s***S*, where *N* is a number from 0 to 15, *C* is a number from 0 to 1, and *S* is a number from 0 to 31, is a character-special file with device number $0x8005+(N<<10)+(C<<9)+(S<<4)$ (major device number $128+N*4+C*2+S/16$, minor device number $(S\%16)*16+5$). This device is slot number *S* of the cioc-data-device located on CIOC number *N* channel number *C*.

A device with the name **/dev/cioctl/u***N*, where *N* is a number from 0 to 15 is a character-special file with device number $0x8014+(N<<10)$ (major device number $128+N*4$, minor device number 20). This device is the cioc-control-device located on CIOC number *N*.

This page is intentionally left blank

NAME

clone – open any minor device on a STREAMS driver

DESCRIPTION

clone is a STREAMS software driver that finds and opens an unused minor device on another STREAMS driver. The minor device passed to *clone* during the open is interpreted as the major device number of another STREAMS driver for which an unused minor device is to be obtained. Each such open results in a separate *stream* to a previously unused minor device.

The *clone* driver consists solely of an open function. This open function performs all of the necessary work so that subsequent system calls (including *close(2)*) require no further involvement of *clone*.

clone will generate an ENXIO error, without opening the device, if the minor device number provided does not correspond to a valid major device, or if the driver indicated is not a STREAMS driver.

CAVEATS

Multiple opens of the same minor device cannot be done through the *clone* interface. Executing *stat(2)* on the file system node for a cloned device yields a different result from executing *fstat(2)* using a file descriptor obtained from opening the node.

SEE ALSO

log(7).
STREAMS Programmer's Guide.

This page is intentionally left blank

NAME

disk – disks and tapes

SYNOPSIS

/dev/dsk/u#c#s#
/dev/dsk/u#c#

/dev/flop
/dev/miniflop
/dev/stream
:
etc.

DESCRIPTION

Connected to the Supermax computer are a number of disks. Typically, such disks are either floppy disks, magnetic tapes, or hard disks (Winchester disks). Hard disks are normally partitioned into smaller so-called sub-disks [see *chlds*(1M) and *l_disk*(2)].

The user sees only so-called *logical disks*. A logical disk is either a removable disk (floppy disk, streamer tape) or a sub-disk on a hard disk.

A device with the name **/dev/dsk/u***N***c***C***s***S*, where *N* is a number from 0 to 15, *C* is a number from 0 to 63, and *S* is a number from 0 to 16, is a block-special file with device number 0x0000+(*N*<<10)+(*C*<<4)+*S* (major device number *N**4+*C*/16, minor device number (*C*%16)*16+*S*). This device is sub-disk number *S* of the disk connected to CPU number *N* (typically, a DIOC) channel number *C*.

The device name **/dev/dsk/u***N***c***C* is used instead of **/dev/dsk/u***N***c***C***s***S*, for disks that are not partitioned into sub-disks.

It is customary to create links with mnemonic names such as, for example, **/dev/flop** and **/dev/stream** to the relevant disks.

The channel numbers are likely to change in future releases of the Supermax Operating System, but at present they are:

1:	First 1MB 8″ floppy
2:	Second 1MB 8″ floppy
3:	First 560KB 5¼″ floppy
4:	Second 560KB 5¼″ floppy
5:	First IBM compatible 8″ floppy
6:	Second IBM compatible 8″ floppy
7:	Streamer tape
8:	First hard disk on first controller
9:	Second hard disk on first controller
10:	First hard disk on second controller
11:	Second hard disk on second controller
12:	First hard disk on third controller
13:	First hard disk on fourth controller
14:	First hard disk on fifth controller
15:	First hard disk on sixth controller
16:	Magtape (low density)
17:	Magtape (high density)

The following *ioctl*(2) requests can be used with disks. The symbolic names for the requests can be found in the **<sys/diskio.h>** and **<sys/ioctl.h>** header files.

W_CHECK
: This request turns read-after-write check off and on. The argument *arg* is either 0, 1 or 2. If *arg* is 0, read-after-write check is turned off (the default setting). If *arg* is 1, read-after-write check is turned on. If *arg* is 2, read-after-write check is left unchanged. The *ioctl*(2) function will in all cases return the old value of the read-after-write check bit. The calling process must be super-user in order to perform this system call.

D_CACHE
: This request turns the disk cache off and on. The argument *arg* is either 0, 1 or 2. If *arg* is 0, the disk cache is turned off (the default setting for swap disks). If *arg* is 1, the disk cache is turned on (the default setting for disks that are not swap disks). If *arg* is 2, the disk cache is left unchanged. The

ioctl(2) function will in all cases return the old value of the the disk cache bit. The calling process must be super-user in order to perform this system call.

DISK_FORMAT	This request causes the disk to be formatted. The argument *arg* is ignored. Only floppy disks can be formatted in this way. The calling process must be super-user in order to perform this system call.
DYN_STAT	This request returns the value of the *deleted data mark* on IBM-compatible floppy disks. The argument *arg* is ignored. If the disk is an IBM-compatible floppy disk and the last read-command read a sector with a *deleted data mark*, the *ioctl*(2) function will return 1, otherwise it will return 0.
TAPE_EOF	This request causes an end-of-file mark to be written to a tape.
TAPE_RWIND	This request causes a tape to be rewound.
TAPE_RETENSION	This request causes a retension operation on a streamer tape.

This page is intentionally left blank

NAME

error – the Operating System error device

SYNOPSIS

#include <sys/errorio.h>

/dev/error

DESCRIPTION

The device with the name **/dev/error** is a character-special file with device number 0x8004 (major device number 128, minor device number 4).

This is a read-only device, from which records of a fixed length can be read. These records give information about events that occur in the operating system and which the system administrator should know about.

The records read have the structure *errrec* defined in the **<sys/errorio.h>** header file. Read requests for fewer bytes than the length of such a record are illegal. A read request will never return more than one record at a time.

Reading is destructive, once a record has been read, it cannot be read again; and positioning within **/dev/error** is meaningless.

If the operating system has no error information available when the read command is issued, the read will pause until information becomes available, unless the O_NDELAY [see *open*(2) and *fcntl*(2)] flag is specified.

SEE ALSO

errlog(1M).

This page is intentionally left blank

NAME

kmem – the kernel memory devices

SYNOPSIS

/dev/kmem

/dev/kmem0, /dev/kmem1, /dev/kmem2, ..., /dev/kmem14, /dev/kmem15

DESCRIPTION

The device with the name **/dev/kmem** is a character-special file with device number 0x8012 (major device number 128, minor device number 18).

From position 0 and up this device refers to the kernel memory of the MCU on which the calling process is executing. From position 0xe00000 and up this device refers to kernel memory that is common to all MCUs. The structure *compart*, defined in the **<sys/compart.h>** header file, describes the data located from position 0xe00000 and up.

Data cannot be written to **/dev/kmem**.

A device with the name **/dev/kmem***N*, where *N* is a number from 0 to 15 is a character special file with device number 0x8002+(*N*<<10) (major device number 128+*N**4, minor device number 2).

From position 0 and up this device refers to the kernel memory of CPU number *N*.

Data can generally not be written to **/dev/kmem***N*.

SEE ALSO

smos_var(2).

This page is intentionally left blank

NAME

null – the null device

SYNOPSIS

/dev/null

DESCRIPTION

The null device has the name **/dev/null**.

The null device is a character special file with device number 0x8001 (major device number 128, minor device number 1).

Data written to the null device is discarded.

Read operations from the null device always return 0 bytes.

This page is intentionally left blank

NAME

print – printers

SYNOPSIS

/dev/prt/u#c#
/dev/prt/u#c#a

/dev/print#
/dev/lp

DESCRIPTION

A device with the name **/dev/prt/u***N***c***C*, where *N* is a number from 0 to 15 and *C* is a number from 0 to 63, is a character-special file with device number

$$0x8000 + (N << 10) + (C << 4)$$

(major device number $128 + N*4 + C/16$), minor device number $(C\%16)*16$. This device is the printer connected to CPU number *N* (typically, a SIOC) channel number *C*.

A device with the name **/dev/prt/u***N***c***C***a**, where *N* is a number from 0 to 15 and *C* is a number from 0 to 63, is a character-special file with device number

$$0x8006 + (N << 10) + (C << 4)$$

(major device number $128 + N*4 + C/16$), minor device number $(C\%16)*16 + 6$). This device is the printer associated with the terminal connected to CPU number *N* (typically, a SIOC) channel number *C*.

It is customary to create links with the names **/dev/lp** and **/dev/print#**, where # is some number, to the printers.

Printers differ from terminals in that a printer cannot be opened if another process already has it open.

The *ioctl*(2) commands described in *termio*(7) can be used with printers.

SEE ALSO

term(7), termio(7).

This page is intentionally left blank

NAME

SA – devices administered by System Administration

DESCRIPTION

The files in the directories **/dev/SA** (for block devices) and the **/dev/rSA** (for raw devices) are used by System Administration to access the devices on which it operates. For devices that support more than one partition (like disks) the **/dev/(r)SA** entry is linked to the partition that spans the entire device. Not all **/dev/(r)SA** entries are used by all System Administration commands.

FILES

/dev/SA
/dev/rSA

SEE ALSO

sysadm(1).

This page is intentionally left blank

235

NAME

sp − STREAMS pipe

SYNOPSIS

/dev/sp

DESCRIPTION

sp is a STREAMS pipe. It can be used to provide a bi-directional pipe between two processes. The pipe is implemented used the STREAMS mechanism.

/dev/sp is a clone device with device number 0xc0c1 (major device number 192, minor device number 193). This clone device selects the first unopen streams pipe device.

In order to create a pipe between two file descriptors, a process must open **/dev/sp** twice (yielding file descriptors *fd1* and *fd2*) and then execute the following code:

```
#include <sys/types.h>
#include <stropts.h>
   :
   :
struct strfdinsert fdi;
long dummy;
   :
   :
fdi.databuf.maxlen = fdi.databuf.len = -1;
fdi.databuf.buf = 0;
fdi.ctlbuf.maxlen = fdi.ctlbuf.len = 4;
fdi.ctlbuf.buf = (caddr_t)&dummy;
fdi.offset = 0;
fdi.fildes = fd2;
fdi.flags = 0;

ioctl (fd1, I_FDINSERT, &fdi);
```

After this, *fd1* and *fd2* will be the two ends of a bi-directional pipe.

SEE ALSO

STREAMS Primer

pipe(2), streamio(7).

NAME

streamio – STREAMS ioctl commands

SYNOPSIS

#include <stropts.h>
int ioctl (fildes, command, arg)
int fildes, command;

DESCRIPTION

STREAMS [see *intro*(2)] ioctl commands are a subset of *ioctl*(2) system calls which perform a variety of control functions on *streams*. The arguments *command* and *arg* are passed to the file designated by *fildes* and are interpreted by the *stream head*. Certain combinations of these arguments may be passed to a module or driver in the *stream*.

fildes is an open file descriptor that refers to a *stream*. *command* determines the control function to be performed as described below. *arg* represents additional information that is needed by this command. The type of *arg* depends upon the command, but it is generally an integer or a pointer to a *command*-specific data structure.

Since these STREAMS commands are a subset of *ioctl*, they are subject to the errors described there. In addition to those errors, the call will fail with *errno* set to EINVAL, without processing a control function, if the *stream* referenced by *fildes* is linked below a multiplexor, or if *command* is not a valid value for a *stream*.

Also, as described in *ioctl*, STREAMS modules and drivers can detect errors. In this case, the module or driver sends an error message to the *stream head* containing an error value. This causes subsequent system calls to fail with *errno* set to this value.

COMMAND FUNCTIONS

The following *ioctl* commands, with error values indicated, are applicable to all STREAMS files:

I_PUSH Pushes the module whose name is pointed to by *arg* onto the top of the current *stream*, just below the *stream head*. It then calls the open routine of the newly-pushed module. On failure, *errno* is set to one of the following values:

[EINVAL]	Invalid module name.
[EFAULT]	*arg* points outside the allocated address space.
[ENXIO]	Open routine of new module failed.
[ENXIO]	Hangup received on *fildes*.

I_POP Removes the module just below the *stream head* of the *stream* pointed to by *fildes*. *arg* should be 0 in an I_POP request. On failure, *errno* is set to one of the following values:

[EINVAL]	No module present in the *stream*.
[ENXIO]	Hangup received on *fildes*.

I_LOOK Retrieves the name of the module just below the *stream head* of the *stream* pointed to by *fildes*, and places it in a null terminated character string pointed at by *arg*. The buffer pointed to by *arg* should be at least FMNAMESZ+1 bytes long. An `[#include <sys/conf.h>]` declaration is required. On failure, *errno* is set to one of the following values:

[EFAULT]	*arg* points outside the allocated address space.
[EINVAL]	No module present in *stream*.

I_FLUSH This request flushes all input and/or output queues, depending on the value of *arg*. Legal *arg* values are:

FLUSHR	Flush read queues.
FLUSHW	Flush write queues.
FLUSHRW	Flush read and write queues.

On failure, *errno* is set to one of the following values:

[ENOSR]	Unable to allocate buffers for flush message due to insufficient STREAMS memory resources.
[EINVAL]	Invalid *arg* value.
[ENXIO]	Hangup received on *fildes*.

I_SETSIG Informs the *stream head* that the user wishes the kernel to issue the SIGPOLL signal [see *signal*(2) and *sigset*(2)] when a particular event has occurred on the *stream* associated with *fildes*. I_SETSIG supports an asynchronous processing capability in STREAMS. The value of *arg* is a bitmask that specifies the events for which the user should be signaled. It is the bitwise-OR of any combination of the following constants:

S_INPUT	A non-priority message has arrived on a *stream head* read queue, and no other messages existed on that queue before this message was placed there. This is set even if the message is of zero length.
S_HIPRI	A priority message is present on the *stream head* read queue. This is set even if the message is of zero length.
S_OUTPUT	The write queue just below the *stream head* is no longer full. This notifies the user that there

is room on the queue for sending (or writing) data downstream.

S_MSG	A STREAMS signal message that contains the SIGPOLL signal has reached the front of the *stream head* read queue.

A user process may choose to be signaled only of priority messages by setting the *arg* bitmask to the value S_HIPRI.

Processes that wish to receive SIGPOLL signals must explicitly register to receive them using I_SETSIG. If several processes register to receive this signal for the same event on the same Stream, each process will be signaled when the event occurs.

If the value of *arg* is zero, the calling process will be unregistered and will not receive further SIGPOLL signals. On failure, *errno* is set to one of the following values:

[EINVAL]	*arg* value is invalid or *arg* is zero and process is not registered to receive the SIGPOLL signal.
[EAGAIN]	Allocation of a data structure to store the signal request failed.

I_GETSIG Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask pointed to by *arg*, where the events are those specified in the description of I_SETSIG above. On failure, *errno* is set to one of the following values:

[EINVAL]	Process not registered to receive the SIGPOLL signal.

240

[EFAULT] *arg* points outside the allocated address space.

I_FIND Compares the names of all modules currently present in the *stream* to the name pointed to by *arg*, and returns 1 if the named module is present in the *stream*. It returns 0 if the named module is not present. On failure, *errno* is set to one of the following values:

[EFAULT] *arg* points outside the allocated address space.

[EINVAL] *arg* does not contain a valid module name.

I_PEEK Allows a user to retrieve the information in the first message on the *stream head* read queue without taking the message off the queue. *arg* points to a *strpeek* structure which contains the following members:

```
struct strbuf   ctlbuf;
struct strbuf   databuf;
long            flags;
```

The *maxlen* field in the *ctlbuf* and *databuf* *strbuf* structures [see *getmsg*(2)] must be set to the number of bytes of control information and/or data information, respectively, to retrieve. If the user sets *flags* to RS_HIPRI, I_PEEK will only look for a priority message on the *stream head* read queue.

I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the *stream head* read queue, or if the RS_HIPRI flag was set in *flags* and a priority message was not present on the *stream head* read queue. It does not wait for a message to arrive. On return, *ctlbuf* specifies information in the control buffer, *databuf* specifies information in the data buffer, and *flags* contains the value 0 or RS_HIPRI. On failure, *errno* is set to the following value:

[EFAULT]	*arg* points, or the buffer area specified in *ctlbuf* or *databuf* is, outside the allocated address space.
[EBADMSG]	Queued message to be read is not valid for I_PEEK

I_SRDOPT Sets the read mode using the value of the argument *arg*. Legal *arg* values are:

RNORM	Byte-stream mode, the default.
RMSGD	Message-discard mode.
RMSGN	Message-nondiscard mode.

Read modes are described in *read*(2). On failure, *errno* is set to the following value:

[EINVAL]	*arg* is not one of the above legal values.

I_GRDOPT Returns the current read mode setting in an *int* pointed to by the argument *arg*. Read modes are described in *read*(2). On failure, *errno* is set to the following value:

[EFAULT]	*arg* points outside the allocated address space.

I_NREAD Counts the number of data bytes in data blocks in the first message on the *stream head* read queue, and places this value in the location pointed to by *arg*. The return value for the command is the number of messages on the *stream head* read queue. For example, if zero is returned in *arg*, but the *ioctl* return value is greater than zero, this indicates that a zero-length message is next on the queue. On failure, *errno* is set to the following value:

[EFAULT] *arg* points outside the allocated address space.

I_FDINSERT Creates a message from user specified buffer(s), adds information about another *stream* and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below.

arg points to a *strfdinsert* structure which contains the following members:

```
struct strbuf   ctlbuf;
struct strbuf   databuf;
long            flags;
int             fildes;
int             offset;
```

The *len* field in the *ctlbuf strbuf* structure [see *putmsg*(2)] must be set to the size of a pointer plus the number of bytes of control information to be sent with the message. *fildes* in the *strfdinsert* structure specifies the file descriptor of the other *stream*. *offset*, which must be word-aligned, specifies the number of bytes beyond the beginning of the control buffer where I_FDINSERT will store a pointer. This pointer will be the address of the read queue structure of the driver for the *stream* corresponding to *fildes* in the *strfdinsert* structure. The *len* field in the *databuf strbuf* structure must be set to the number of bytes of data information to be sent with the message or zero if no data part is to be sent.

flags specifies the type of message to be created. A non-priority message is created if *flags* is set to 0, and a priority message is created if *flags* is set to RS_HIPRI. For non-priority messages, I_FDINSERT will block if the *stream* write queue is full due to internal flow control conditions. For priority messages, I_FDINSERT does not block on this condition. For non-priority messages, I_FDINSERT does not block when the write queue is full and O_NDELAY is set. Instead, it fails and sets *errno* to EAGAIN.

I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the *stream*, regardless of priority or whether O_NDELAY has been specified. No partial message is sent. On failure, *errno* is set to one of the following values:

[EAGAIN]	A non-priority message was specified, the O_NDELAY flag is set, and the *stream* write queue is full due to internal flow control conditions.
[ENOSR]	Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.
[EFAULT]	*arg* points, or the buffer area specified in *ctlbuf* or *databuf* is, outside the allocated address space.
[EINVAL]	One of the following: *fildes* in the *strfdinsert* structure is not a valid, open *stream* file descriptor; the size of a pointer plus *offset* is greater than the *len* field for the buffer specified through *ctlptr*; *offset* does not specify a properly-

aligned location in the data buffer; an undefined value is stored in *flags*.

[ENXIO] Hangup received on *fildes* of the *ioctl* call or *fildes* in the *strfdinsert* structure.

[ERANGE] The *len* field for the buffer specified through *databuf* does not fall within the range specified by the maximum and minimum packet sizes of the topmost *stream* module, or the *len* field for the buffer specified through *databuf* is larger than the maximum configured size of the data part of a message, or the *len* field for the buffer specified through *ctlbuf* is larger than the maximum configured size of the control part of a message.

I_FDINSERT can also fail if an error message was received by the *stream head* of the *stream* corresponding to *fildes* in the *strfdinsert* structure. In this case, *errno* will be set to the value in the message.

I_STR Constructs an internal STREAMS ioctl message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send user *ioctl* requests to downstream modules and drivers. It allows information to be sent with the *ioctl*, and will return to the user any information sent upstream by the downstream recipient. I_STR blocks until the system responds with either a positive or negative acknowledgement message, or until the request "times out" after some period of time. If the request times out, it fails with *errno* set to ETIME.

At most, one I_STR can be active on a *stream*. Further I_STR calls will block until the active I_STR completes at the *stream head*. The default timeout interval for these requests is 15 seconds. The O_NDELAY [see *open*(2)] flag has no effect on this call.

To send requests downstream, *arg* must point to a *strioctl* structure which contains the following members:

```
int   ic_cmd;      /* downstream command */
int   ic_timout;   /* ACK/NAK timeout */
int   ic_len;      /* length of data arg */
char *ic_dp;       /* ptr to data arg */
```

ic_cmd is the internal ioctl command intended for a downstream module or driver and *ic_timout* is the number of seconds (-1 = infinite, 0 = use default, >0 = as specified) an I_STR request will wait for acknowledgement before timing out. *ic_len* is the number of bytes in the data argument and *ic_dp* is a pointer to the data argument. The *ic_len* field has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the user (the buffer pointed to by *ic_dp* should be large enough to contain the maximum amount of data that any module or the driver in the *stream* can return).

The *stream head* will convert the information pointed to by the *strioctl* structure to an internal ioctl command message and send it downstream. On failure, *errno* is set to one of the following values:

[ENOSR]	Unable to allocate buffers for the *ioctl* message due to insufficient STREAMS memory resources.
[EFAULT]	*arg* points, or the buffer area specified by *ic_dp* and *ic_len* (separately for data sent and data returned) is, outside the allocated address space.
[EINVAL]	*ic_len* is less than 0 or *ic_len* is larger than the maximum configured size of the data part of a message or *ic_timout* is less than −1.
[ENXIO]	Hangup received on *fildes*.
[ETIME]	A downstream *ioctl* timed out before acknowledgement was received.

An I_STR can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the *stream head*. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the ioctl command sent downstream fails. For these cases, I_STR will fail with *errno* set to the value in the message.

I_SENDFD Requests the *stream* associated with *fildes* to send a message, containing a file pointer, to the *stream head* at the other end of a *stream* pipe. The file pointer corresponds to *arg*, which must be an integer file descriptor.

I_SENDFD converts *arg* into the corresponding system file pointer. It allocates a message block and inserts the file pointer in the block. The user id and group id associated with the sending process are also inserted. This message is placed directly on the read queue [see *intro*(2)] of the *stream head* at the other end of the *stream* pipe to which it is connected. On failure, *errno* is set to one of the following values:

[EAGAIN]	The sending *stream* is unable to allocate a message block to contain the file pointer.
[EAGAIN]	The read queue of the receiving *stream head* is full and cannot accept the message sent by I_SENDFD.
[EBADF]	*arg* is not a valid, open file descriptor.
[EINVAL]	*fildes* is not connected to a *stream* pipe.
[ENXIO]	Hangup received on *fildes*.

I_RECVFD Retrieves the file descriptor associated with the message sent by an I_SENDFD *ioctl* over a *stream* pipe. *arg* is a pointer to a data buffer large enough to hold an *strrecvfd* data structure containing the following members:

```
int fd;
unsigned short uid;
unsigned short gid;
char fill[8];
```

fd is an integer file descriptor. *uid* and *gid* are the user id and group id, respectively, of the sending *stream*.

If O_NDELAY is not set [see *open*(2)], I_RECVFD will block until a message is present at the *stream head*. If O_NDELAY is set, I_RECVFD will fail with *errno* set to EAGAIN if no message is present at the *stream head*.

If the message at the *stream head* is a message sent by an I_SENDFD, a new user file descriptor is allocated for the file pointer contained in the message. The new file descriptor is placed in the *fd* field of the *strrecvfd* structure. The structure is copied into the user data buffer pointed to by *arg*. On failure, *errno* is set to one of the following values:

[EAGAIN]	A message was not present at the *stream head* read queue, and the O_NDELAY flag is set.
[EBADMSG]	The message at the *stream head* read queue was not a message containing a passed file descriptor.
[EFAULT]	*arg* points outside the allocated address space.
[EMFILE]	NOFILES file descriptors are currently open.
[ENXIO]	Hangup received on *fildes*.

The following two commands are used for connecting and disconnecting multiplexed STREAMS configurations.

I_LINK	Connects two *streams*, where *fildes* is the file descriptor of the *stream* connected to the multiplexing driver, and *arg* is the file descriptor of the *stream* connected to another driver. The *stream* designated by *arg* gets connected below the multiplexing driver. I_LINK requires the multiplexing driver to send an acknowledgement message to the *stream head* regarding the linking operation. This call returns a multiplexor ID number (an identifier used to disconnect the multiplexor, see I_UNLINK) on success, and a −1 on failure. On failure, *errno* is set to one of the following values:

[ENXIO]	Hangup received on *fildes*.
[ETIME]	Time out before acknowledgement message was received at *stream head*.
[EAGAIN]	Temporarily unable to allocate storage to perform the I_LINK.
[ENOSR]	Unable to allocate storage to perform the I_LINK due to insufficient STREAMS memory resources.
[EBADF]	*arg* is not a valid, open file descriptor.
[EINVAL]	*fildes stream* does not support multiplexing.
[EINVAL]	*arg* is not a *stream*, or is already linked under a multiplexor.
[EINVAL]	The specified link operation would cause a "cycle" in the resulting configuration; that is, if a given *stream head* is linked into a multiplexing configuration in more than one place.

An I_LINK can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the *stream head* of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_LINK will fail with *errno* set to the value in the message.

I_UNLINK Disconnects the two *streams* specified by *fildes* and *arg*. *fildes* is the file descriptor of the *stream* connected to the multiplexing driver. *fildes* must correspond to the *stream* on which the *ioctl* I_LINK command was issued to link the *stream* below the multiplexing driver. *arg* is the multiplexor ID number that was returned by the I_LINK. If *arg* is -1, then all Streams which were linked to *fildes* are disconnected. As in I_LINK, this command requires the multiplexing driver to acknowledge the unlink. On failure, *errno* is set to one of the following values:

[ENXIO]	Hangup received on *fildes*.
[ETIME]	Time out before acknowledgement message was received at *stream head*.
[ENOSR]	Unable to allocate storage to perform the I_UNLINK due to insufficient STREAMS memory resources.
[EINVAL]	*arg* is an invalid multiplexor ID number or *fildes* is not the *stream* on which the I_LINK that returned *arg* was performed.

An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the *stream head* of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK will fail with *errno* set to the value in the message.

SEE ALSO

close(2), fcntl(2), getmsg(2), intro(2&3), ioctl(2), open(2), poll(2), putmsg(2), read(2), signal(2), sigset(2), write(2).

STREAMS Programmer's Guide.

STREAMS Primer.

DIAGNOSTICS

Unless specified otherwise above, the return value from *ioctl* is 0 upon success and −1 upon failure with *errno* set as indicated.

NAME

term – terminals

SYNOPSIS

/dev/term/u#c#w#
/dev/term/u#c#

/dev/tty#
/dev/console

DESCRIPTION

A device with the name **/dev/term/u***N***c***C***w***W*, where *N* is a number from 0 to 15, *C* is a number from 0 to 63, and *W* is a number from 1 to 6, is a character-special file with device number

$$0x4000 + (N << 10) + (C << 4) + W$$

(major device number $64 + N*4 + C/16$, minor device number $(C\%16)*16 + W$). This device is window number *W* of the terminal connected to CPU number *N* (typically, a SIOC) channel number *C*.

A device with the name **/dev/term/u***N***c***C*, where *N* is a number from 0 to 15 and *C* is a number from 0 to 63, is linked to the file **/dev/term/u***N***c***C***w1**.

It is customary to create links with the names **/dev/console** and **/dev/tty#**, where # is some number, to the terminals.

The *ioctl*(2) commands described in *termio*(7) can be used with terminals.

SEE ALSO

print(7), termio(7), tty(7).

This page is intentionally left blank

NAME

termio − general terminal interface

DESCRIPTION

All of the terminal and printer ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of this interface. The term *terminal* will be used as a generic term for terminals, printers or whatever else is connected to the ports. The discussion below deals mainly with the serial ports located on the SIOC modules of the Supermax Computer; but most of what is said also applies to other terminal and printer ports on the computer.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open terminal files; they are opened by *getty*(1M) and become a user's standard input, output, and error files. The very first opening 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, interrupt, and hangup 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. Terminals may operate in one of a number of *line disciplines*. Line disciplines 0 and 1 are the most commonly used ones. What is said below applies to both line disciplines, except where the opposite is explicitly stated.

Character input − line discipline 0

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, the following characters are ignored.

Normally, terminal input is processed in units of lines. A line is delimited by a *New-line* (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 # 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 *New-line* 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:

Interrupt (Control-C or ASCII ETX) 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). Occasionally the rubout-character (ASCII DEL) is used instead of control-C.

Quit (Control-| or ASCII FS) 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.

Attention	(Control-B or ASCII STX) generates an *attention* signal which is sent to processes that have the terminal as their standard input device. See *signal*(2).
Switch	(Control-Z or ASCII SUB) is used by the job control facility, *shl*(1), to change the current layer to the control layer.
Erase	(#) erases the preceding character. It will not erase beyond the start of a line, as delimited by a *New-line*, *End-of-file*, or *End-of-line* character.
Kill	(@) deletes the entire line, as delimited by a *New-line*, *End-of-file*, or *End-of-line* character.
End-of-file	(Control-D or ASCII EOT) 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 *New-line*, and the *End-of-file* is discarded. Thus, if there are no characters waiting, which is to say the *End-of-file* occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.
New-line	(ASCII LF) is the normal line delimiter. It can not be changed or escaped.
End-of-line	(ASCII NUL) is an additional line delimiter, like *New-line*. 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 *Interrupt*, *Quit*, *Attention*, *Switch*, *Erase*, *Kill*, *End-of-file*, and *End-of-line* may be changed to suit individual tastes. The *Erase*, *Kill*, and *End-of-file* characters may be escaped by a preceding \ character, in which case no special function is done.

Character input – line discipline 1

When the terminal is doing so-called *canonical input*, characters typed when a read-command is not in effect are normally lost. The only exception is the type-ahead facility of the *edit*(2) system call.

Normally, terminal input is processed in units of lines. A line is delimited by a *New-line* (ASCII LF) character, an *End-of-file* (ASCII EOT), a *Carriage-return* (ASCII CR), or a non-editing function key. 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, line editing is normally done as specified below.

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

Interrupt (Control-C or ASCII ETX) 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). Occasionally the rubout-character (ASCII DEL) is used instead of control-C.

Quit	(Control-| or ASCII FS) 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.
Attention	(Control-B or ASCII STX) generates an *attention* signal which is sent to processes that have the terminal as their standard input device. See *signal*(2).
Switch	(Control-Z or ASCII SUB) is used by job control facilities, such as *shl*(1) and *ds*(1), to change the current layer to the control layer.
Insert-character	(Function key F6, value 0x01 0x45) inserts a space at the cursor position.
Delete-character	(Function key SHIFT/F6, value 0x01 0x65) deletes the character at the cursor position.
Go-to-end-of-line	(Function key F7, value 0x01 0x46) moves the cursor to the end of the typed line.
Go-to-beginning-of-line	(Function key SHIFT/F7, value 0x01 0x66) moves the cursor to the beginning of the line.
Erase-to-end-of-line	(Function key F8, value 0x01 0x47) erases the line from the cursor position to the end of the line.
Kill	(Function key SHIFT/F8, value 0x01 0x67) erases the entire line.

Toggle-insert-mode
(Key sequence Control-A I I, function key value 0x01 0x39) toggles insert mode.

Right-arrow (Control-L or ASCII FF) moves the cursor one position right.

Left-arrow (Control-H or ASCII BS) moves the cursor one position left.

Erase (Rubout or ASCII DEL) erases the character preceding the cursor. When line editing is in insert mode the erasure is done as "Backspace, Delete character". When line editing is not in insert mode the erasure is done as "Backspace, Space, Backspace".

End-of-file (Control-D or ASCII EOT) 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 *New-line*, and the *End-of-file* is discarded. Thus, if there are no characters waiting, which is to say the *End-of-file* occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.

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

Return (ASCII CR) is an alternative line delimiter. It can not be changed or escaped.

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 *Interrupt*, *Quit*, *Attention*, and *Switch* may be changed to suit individual tastes.

Both line disciplines

When the carrier signal from the data-set drops, a *hang-up* 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 hang-up 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 16

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 symbolic names and offsets in *c_cc* for each function are as follows:

Interrupt	VINTR	0
Quit	VQUIT	1
Erase	VERASE	2
Kill	VKILL	3
End-of-file	VEOF	4
End-of-line	VEOL	5
Switch	VSWTCH	7
Attention	VATT	9
VT bit	VTBIT	10

The *Erase*, *Kill*, *End-of-file*, and *End-of-line* characters specified in the *c_cc* array apply only to line discipline 0. They cannot be changed in line discipline 1.

The "VT bit" character is really not a character. Only the least significant two bits are relevant. They control whether the Virtual Terminal Interface is enabled or not (see the *Supermax Virtual Terminal Guide*). If the least significant bit (value 1) of this field is 1, Virtual Terminal translation on input is enabled. If the second least significant bit (value 2) of this field is 1, Virtual Terminal translation on output is enabled.

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 *New-line* to *Carriage-return* on input.
IGNCR	*	0000200	Ignore *Carriage-return*.
ICRNL	*	0000400	Map *Carriage-return* to *New-line* 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.

The flags marked with a minus (–) have no effect in the Supermax Operating System, but are included for reasons of compatibility. The flags marked with an asterisk (*) have no effect in line discipline 1. The flags marked with a dagger (†) have no effect on a NIOC (Network Input/Output Controller).

The break condition is not put in the input queue and is therefore not read by any process. If BRKINT is set, the break condition will generate an *interrupt* signal and flush both the input and output queues.

If PARMRK is set and ISTRIP is not set, a valid character of 0377 is read as 0377, 0377.

Input parity checking is always disabled. Characters with parity errors are never ignored. Characters with framing errors are always ignored.

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 *New-line* character is translated into a *Carriage-return* character. If IGNCR is set, a received *Carriage-return* character is ignored (not read). Otherwise if ICRNL is set, a received *Carriage-return* character is translated into a *New-line* 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. Otherwise, hardware handshake is enabled.

The initial input control value is that IXON, and IXOFF are set, and all others are 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 *New-line* to *Carriage-return*—*New-line* on output.
OCRNL		0000010	Map *Carriage-return* to *New-line* on output.
ONOCR		0000020	No *Carriage-return* output at column 0.
ONLRET		0000040	*New-line* performs *Carriage-return* 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	

The flags marked with a minus (–) have no effect in the Supermax Operating System, but are included for reasons of compatibility. Fill characters for delay purposes may, however, be sent by the Virtual Terminal Interface.

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

If ONLCR is set, the *New-line* character is transmitted as the *Carriage-return*–*New-line* character pair. If OCRNL is set, the *Carriage-return* character is transmitted as the *New-line* character. If ONOCR is set, no *Carriage-return* character is transmitted when at column 0 (first position). If ONLRET is set, the *New-line* character is assumed to do the *Carriage-return* function; the column pointer will be set to 0 and the delays specified for *Carriage-return* will be used. Otherwise the *New-line* 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 *Carriage-return* character is actually transmitted.

If TAB3 is set, tab characters are expanded to spaces, otherwise, tab characters are output directly.

The initial output control value is that OPOST, ONLCR, and TAB3 are set, and all others are clear.

The *c_cflag* field describes the hardware control of the terminal:

CBAUD	†	0000017	Baud rate:
B0	†	0	Hang up
B50	†	0000001	50 baud
B75	†	0000002	75 baud
B110	†	0000003	110 baud
B134	†	0000004	134 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
B19200	†	0000016	19200 baud
B38400	†	0000017	38400 baud
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 flag marked with a minus (–) has no effect in the Supermax Operating System, but are included for reasons of compatibility. The flags marked with a dagger (†) have no effect on a NIOC (Network Input/Output Controller).

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.

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 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 is specified by the *chhw*(1M) program.

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions:

ISIG		0000001	Enable signals.
ICANON		0000002	Canonical input (line editing).
XCASE	–	0000004	Canonical upper/lower presentation.
ECHO		0000010	Enable echo.
ECHOE		0000020	Echo erase character as backspace-space-backspace.
ECHOK	*	0000040	Echo *New-line* after kill character.
ECHONL		0000100	Echo *New-line*.
NOFLSH		0000200	Disable flush after *interrupt* or *quit*.

The flag marked with a minus (–) has no effect in the Supermax Operating System, but is included for reasons of compatibility. The flag marked with an asterisk (*) has no effect in line discipline 1.

If ISIG is set, each input character is checked against the special control characters *Interrupt*, *Quit*, *Attention*, and *Switch*. 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 (for example, ASCII NUL).

If ICANON is set, canonical processing is enabled. This enables the line editing functions, and the assembly of input characters into lines delimited by *New-line*, *End-of-file*, and *End-of-line*. If ICANON is not set, read requests are satisfied directly from the input queue, and characters will not be echoed. A read will not be satisfied until at least MIN characters have been received or the timeout value TIME has expired between characters. This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN and TIME values are stored in the position for the *End-of-file* and *End-of-line* characters, respectively. For this purpose the symbolic names VMIN and VTIME are provided as synonyms for VEOF an VEOL, respectively. The time value represents tenths of seconds.

When ICANON is set, the following echo functions are possible. If ECHO is set, characters are echoed as received. 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 *New-line* 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 *New-line* 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 *End-of-file* character is not echoed. Because EOT is the default *End-of-file* character, this prevents terminals that respond to EOT from hanging up.

In line discipline 1, *End-of-file* is echoed as the *New-line* character.

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 that ISIG, ICANON, ECHO, ECHOE, ECHONL, and NOFLSH are set, and all others are clear.

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

ioctl (fildes, command, arg)

The legal commands are:

TCGETA	With this system call, *arg* is of type *struct termio* *. The system call gets the parameters associated with the terminal and stores them in the structure referenced by *arg*.
TCSETA	With this system call, *arg* is of type *struct termio* *. The system call sets the parameters associated with the terminal from the structure referenced by *arg*. The change is immediate.
TCSETAW	With this system call, *arg* is of type *struct termio* *. The system call is identical to TCSETA, except that it waits for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.
TCSETAF	With this system call, *arg* is of type *struct termio* *. The system call is identical to TCSETA, except that it waits for the output to drain, then flushes the input queue and sets the new parameters.

NOTE: In standard UNIX the *c_cc* array of the *termio* structure has only 8 elements. In order to maintain compatibility with standard UNIX, the above-mentioned *ioctl* calls will read or set only the first 8 characters of the *c_cc* array. If all 16 characters are to be read or set, the *ioctl* call must be preceded by a *set_parm*(2) system call, whose first parameter is different from −1.

TCSBRK

With this system call, *arg* is of type *int*. The system call waits for the output to drain. If *arg* is 0, the call then sends a break (zero bits for 0.25 seconds).

TCXONC

With this system call, *arg* is of type *int*. The system call provides start/stop control. If *arg* is 0, output is suspended; if 1, suspended output is restarted.

TCFLSH

With this system call, *arg* is of type *int*. If *arg* is 0, the input queue is flushed; if 1, the output queue is flushed; if 2, both the input and output queues are flushed.

FIONREAD

With this system call, *arg* is of type *int* ∗. The system call stores the number of characters currently in the terminal's input buffer into the location pointed to by *arg*.

SET_DI

With this system call, *arg* is ignored. The system call puts the terminal in *direct input mode*. This is the same as turning off the ICANON flag, with the exception that when the process terminates, the operating system automatically removes the direct input mode.

CLEAR_DI

With this system call, *arg* is ignored. The system call removes the direct input mode set by the SET_DI call above.

DYN_STAT

With this system call, *arg* is of type *struct fkey* ∗. The structure looks like this:

```
struct fkey {
  unsigned charfk_fkey1;
  unsigned charfk_curoff;
  unsigned shorfk_fkey2;
};
```

The system call stores in the structure pointed to by *arg* various input information: The *fk_fkey1* field will contain the value of the most recently pressed function key. The *fk_fkey2* field will contain the value of the key that terminated the last input statement. If *fk_fkey2*>0x100, the key is a function key. The *fk_curoff* field contains the cursor offset at the end of the last input operation.

The following *ioctl*(2) calls apply only to terminals with more than one window.

SXTIOCLINK
: With this system call, *arg* is of type *int*. The system call initiates windowing. The number of windows is specified by *arg*.

SXTIOCSWTCH
: With this system call, *arg* is of type *int*. The system call switches input to be taken from window number *arg*. This system call can only be issued from window number 0.

SXTIOCWF
: With this system call, *arg* is of type *int*. The system call causes the calling process to wait until window number *arg* becomes the controlling window. Window number 0 becomes the controlling window when the *Switch* character is pressed. Other windows become the controlling window as a result of the SXTIOCSWTCH call above.

FILES

/dev/tty * , /dev/print * , /dev/tty.

SEE ALSO

stty(1), fork(2), ioctl(2), setpgrp(2), signal(2)

NAME

timod – Transport Interface cooperating STREAMS module

DESCRIPTION

timod is a STREAMS module for use with the Transport Interface (TI) functions of the Network Services library. The *timod* module converts a set of *ioctl*(2) calls into STREAMS messages that may be consumed by a transport protocol provider which supports the Transport Interface. This allows a user to initiate certain TI functions as atomic operations.

The *timod* module must be pushed (see *Streams Primer*) onto only a *stream* terminated by a transport protocol provider which supports the TI.

All STREAMS messages, with the exception of the message types generated from the *ioctl* commands described below, will be transparently passed to the neighboring STREAMS module or driver. The messages generated from the following *ioctl* commands are recognized and processed by the *timod* module. The format of the *ioctl* call is:

```
#include <sys/stropts.h>
              -
              -
struct strioctl strioctl;
              -
              -
strioctl.ic_cmd = cmd;
strioctl.ic_timeout = INFTIM;
strioctl.ic_len = size;
strioctl.ic_dp = (char *)buf

ioctl(fildes, I_STR, &strioctl);
```

Where, on issuance, *size* is the size of the appropriate TI message to be sent to the transport provider and on return *size* is the size of the appropriate TI message from the transport provider in response to the issued TI message.

buf is a pointer to a buffer large enough to hold the contents of the appropriate TI messages. The TI message types are defined in <*sys/tihdr.h*>. The possible values for the *cmd* field are:

TI_BIND	Bind an address to the underlying transport protocol provider. The message issued to the TI_BIND *ioctl* is equivalent to the TI message type T_BIND_REQ and the message returned by the successful completion of the *ioctl* is equivalent to the TI message type T_BIND_ACK.
TI_UNBIND	Unbind an address from the underlying transport protocol provider. The message issued to the TI_UNBIND *ioctl* is equivalent to the TI message type T_UNBIND_REQ and the message returned by the successful completion of the *ioctl* is equivalent to the TI message type T_OK_ACK.
TI_GETINFO	Get the TI protocol specific information from the transport protocol provider. The message issued to the TI_GETINFO *ioctl* is equivalent to the TI message type T_INFO_REQ and the message returned by the successful completion of the *ioctl* is equivalent to the TI message type T_INFO_ACK.
TI_OPTMGMT	Get, set or negotiate protocol specific options with the transport protocol provider. The message issued to the TI_OPTMGMT *ioctl* is equivalent to the TI message type T_OPTMGMT_REQ and the message returned by the successful completion of the *ioctl* is equivalent to the TI message type T_OPTMGMT_ACK.

FILES

<sys/timod.h>
<sys/tiuser.h>
<sys/tihdr.h>
<sys/errno.h>

SEE ALSO

tirdwr(7).

STREAMS Primer.

STREAMS Programmer's Guide.

Network Programmer's Guide.

DIAGNOSTICS

If the *ioctl* system call returns with a value greater than 0, the lower 8 bits of the return value will be one of the TI error codes as defined in *<sys/tiuser.h>*. If the TI error is of type TSYSERR, then the next 8 bits of the return value will contain an error as defined in *<sys/errno.h>* (see *intro*(2)).

This page is intentionally left blank

NAME

tirdwr – Transport Interface read/write interface STREAMS module

DESCRIPTION

tirdwr is a STREAMS module that provides an alternate interface to a transport provider which supports the Transport Interface (TI) functions of the Network Services library (see Section 3N). This alternate interface allows a user to communicate with the transport protocol provider using the *read*(2) and *write*(2) system calls. The *putmsg*(2) and *getmsg*(2) system calls may also be used. However, *putmsg* and *getmsg* can only transfer data messages between user and *stream*.

The *tirdwr* module must only be pushed [see I_PUSH in *streamio*(7)] onto a *stream* terminated by a transport protocol provider which supports the TI. After the *tirdwr* module has been pushed onto a *stream*, none of the Transport Interface functions can be used. Subsequent calls to TI functions will cause an error on the *stream*. Once the error is detected, subsequent system calls on the *stream* will return an error with *errno* set to EPROTO.

The following are the actions taken by the *tirdwr* module when pushed on the *stream*, popped [see I_POP in *streamio*(7)] off the *stream*, or when data passes through it.

push – When the module is pushed onto a *stream*, it will check any existing data destined for the user to ensure that only regular data messages are present. It will ignore any messages on the *stream* that relate to process management, such as messages that generate signals to the user processes associated with the *stream*. If any other messages are present, the I_PUSH will return an error with *errno* set to EPROTO.

write – The module will take the following actions on data that originated from a *write* system call:

- All messages with the exception of messages that contain control portions (see the *putmsg* and *getmsg* system calls) will be transparently passed onto the module's downstream neighbor.
- Any zero length data messages will be freed by the module and they will not be passed onto the module's downstream neighbor.
- Any messages with control portions will generate an error, and any further system calls associated with the *stream* will fail with *errno* set to EPROTO.

read – The module will take the following actions on data that originated from the transport protocol provider:

- All messages with the exception of those that contain control portions (see the *putmsg* and *getmsg* system calls) will be transparently passed onto the module's upstream neighbor.
- The action taken on messages with control portions will be as follows:
  + Messages that represent expedited data will generate an error. All further system calls associated with the *stream* will fail with *errno* set to EPROTO.
  + Any data messages with control portions will have the control portions removed from the message prior to passing the message on to the upstream neighbor.
  + Messages that represent an orderly release indication from the transport provider will generate a zero length data message, indicating the end of file, which will

be sent to the reader of the *stream*. The orderly release message itself will be freed by the module.

+ Messages that represent an abortive disconnect indication from the transport provider will cause all further *write* and *putmsg* system calls to fail with *errno* set to ENXIO. All further *read* and *getmsg* system calls will return zero length data (indicating end of file) once all previous data has been read.

+ With the exception of the above rules, all other messages with control portions will generate an error and all further system calls associated with the *stream* will fail with *errno* set to EPROTO.

– Any zero length data messages will be freed by the module and they will not be passed onto the module's upstream neighbor.

pop – When the module is popped off the *stream* or the *stream* is closed, the module will take the following action:

– If an orderly release indication has been previously received, then an orderly release request will be sent to the remote side of the transport connection.

SEE ALSO

getmsg(2), intro(2&3), putmsg(2), read(2), write(2), streamio(7), timod(7).

STREAMS Primer.

STREAMS Programmer's Guide.

Network Programmer's Guide.

This page is intentionally left blank

NAME

tty – controlling terminal

SYNOPSIS

/dev/tty

DESCRIPTION

The device with the name **/dev/tty** is a character-special file with device number 0x8003 (major device number 128, minor device number 3). This device is a synonym for the control terminal, if any, associated with the calling 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, typed output is desired, and it is tiresome to find out what terminal is currently in use.

The *ioctl*(2) commands described in *termio*(7) can be used with this device.

SEE ALSO

print(7), term(7), termio(7).

This page is intentionally left blank

This page will be replaced by the manual pages describing the programs contained in the separately supplied *Local DDE Utilities*.

The programs in *Local DDE Utilities* are not standard supplement, but contains special DDE developed useful utilities.

For more detailed information about the *Local DDE Utilities*, please contact Your Supermax consultant.

Currently available are the following programs:

Stock no.:

30110991	Tar Backup System
30120991	Modem Logon System

This page is intentionally left blank

This page will be replaced by the manual pages describing the programs contained in the separately supplied *Local DDE Subroutines*.

The programs in *Local DDE Subroutines* are not standard supplement, but contains special DDE developed useful subroutines intended to be linked into application programs.

For more detailed information about the *Local DDE Subroutines*, please contact Your Supermax consultant.

Currently available is the following program:

Stock no.:	
30150991	Mag Tape Utilities

285

This page is intentionally left blank

NOTATER

NOTATER