UNISYS

System V Operating System

Programmer's Reference Manual

Volume 3

December 1987

Printed in U S America
UP–13713

Priced Item

This document is intended for software releases based on AT&T Release 3 of UNIX System V or a subsequent release of the System unless otherwise indicated.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Table of Contents

(The following are contained in three volumes.)

1. Commands

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 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, a 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.
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.

The **a.out** shared text executable file produced by *ld*(1) has the magic number in the first field of the UNIX system header. The headers (file header, UNIX system header, and section headers) are loaded at the beginning of the text segment and the text immediately follows the headers in the user address space. The first text address will equal the starting location of the text segment plus the size of the headers, and will vary depending upon the number of section headers in the **a.out** file. The text segment is not writable by the program; if other processes are executing the same **a.out** file, the processes will share a single text segment.

The data segment starts at the next 512K boundary past the last text address. The first data address is determined by the following: If an **a.out** file were split into 8K chunks, one of the chunks would contain both the end of text and the beginning of data. When the core image is created, that chunk will appear twice; once at the end of text and once at the beginning of data (with some unused space in between). The duplicated chunk of text that appears at the beginning of data is never executed; it is duplicated so that the operating system may bring in pieces of the file in multiples of the page size without having to realign the beginning of the data section to a page boundary. Therefore the first data address is the sum of the next segment boundary past the end of text plus the remainder of the last text address divided by 8K. If the last text address is a multiple of 8K no duplication is necessary.

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

An example format of the **filehdr** header is

```
struct filehdr
{
  unsigned short    f_magic;   /* magic number */
  unsigned short    f_nscns;   /* number of sections */
  long              f_timdat;  /* time and 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 */
};
```

UNIX System Header

An example 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

An example 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 */
  unsigned short    s_nreloc;/* # reloc entries */
  unsigned short    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 example 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

An example format of each symbol in the symbol table is

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

struct syment
{
  union                        /* 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 */
    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]
```

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

```
union auxent {
  struct {
      long   x_tagndx;
      union {
            struct {
                  unsigned shortx_lnno;
                  unsigned shortx_size;
            } x_lnsz;
            long   x_fsize;
      } x_misc;
      union {
            struct {
                  long   x_lnnoptr;
                  long   x_endndx;
            }  x_fcn;
            struct {
                  unsigned shortx_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).

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 acct      acctbuf;
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, as modified by text-sharing.

The structure **tacct.h**, 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(2), exec(2), fork(2).
acct(1M) in the *System Administrator's Reference Manual.*
acctcom(1) in the *User's Reference Manual.*

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

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).

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) in the *System Administrator's Reference Manual*.

[This page left blank.]

NAME

core - format of core image file

DESCRIPTION

The UNIX 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 first section of the core image is a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter *usize*, which is defined in **<sys/param.h>**. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the *user* structure of the system, defined in **<sys/user.h>**. Not included in this file are the locations of the registers. These are outlined in **<sys/reg.h>**.

SEE ALSO

sdb(1), setuid(2), signal(2).
crash(1M) in the *System Administrator's Reference Manual.*

[This page 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%11o%6o%11o%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

stat(2).
cpio(1), find(1) in the *User's Reference Manual*.

[This page 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)]. A common structure of a directory entry as given in the include file is:

```
#ifndef DIRSIZ
#define DIRSIZ      14
#endif
struct   direct
{
        ino_t  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 left blank.]

NAME

dirent - file system independent directory entry

SYNOPSIS

#include <sys/dirent.h>
#include <sys/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.

An example *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/sys/dirent.h

SEE ALSO

getdents(2).

[This page 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. The valid magic numbers are given below:

```
#define FBOMAGIC    0560     /* 3B2 and 3B5 computers */
#define N3BMAGIC    0550     /* 3B20 computer */
#define NTVMAGIC    0551     /* 3B20 computer */

#define VAXWRMAGIC  0570     /* VAX writable text segments */
#define VAXROMAGIC  0575     /* VAX r. o. sharable text seg. */
```

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  /* 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/filsys.h>
#include <sys/types.h>
#include <sys/param.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, or 1024-byte long sectors, depending upon your particular machine. Sector 0 is generally unused and is available to contain a bootstrap program or other information.

Sector 1 is the *super-block*. An example 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 */
  ino_t    s_inode[NICINOD];/* free i-node list */
  char     s_flock;        /* lock during free list manip. */
  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*/
  ino_t    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  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 supported: the original 512-byte logical block and the improved 1024-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 1024-byte oriented file system, a block is 1024-bytes or two 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

mount(2), inode(4).
fsck(1M), fsdb(1M), mkfs(1M) in the *System Administrator's Reference Manual.*

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.

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) in the *User's Reference Manual.*

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

NOTE

The name of the file-system-table is machine specific. Other possible names could be **/etc/mountable** or **/etc/mntnodes.**

SEE ALSO

mount(1M), mountall(1M), rmountall(1M) in the *System Administrator's Reference Manual.*

[This page 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* 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* 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*. The flags that *getty* 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* 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* executes *login*(1).

final-flags — These flags take the same values as the *initial-flags* and are set just prior to *getty* executes *login*. 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. The other two 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* 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* 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* can not find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command 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* with the check option to be sure there are no errors.

FILES

/etc/gettydefs

SEE ALSO

ioctl(2).
getty(1M), termio(7) in the *System Administrator's Reference Manual*.
login(1) in the *User's Reference Manual*.

NAME

gps - graphical primitive string, format of graphical files

DESCRIPTION

GPS is a format used to store graphical data. Several routines have been developed to edit and display GPS files on various devices. Also, higher level graphics programs such as *plot* [in *stat*(1G)] and *vtoc* [in *toc*(1G)] produce GPS format output files.

A GPS is composed of five types of graphical data or primitives.

GPS PRIMITIVES

lines The *lines* primitive has a variable number of points from which zero or more connected line segments are produced. The first point given produces a *move* to that location. (A *move* is a relocation of the graphic cursor without drawing.) Successive points produce line segments from the previous point. Parameters are available to set *color*, *weight*, and *style* (see below).

arc The *arc* primitive has a variable number of points to which a curve is fit. The first point produces a *move* to that point. If only two points are included, a line connecting the points will result; if three points a circular arc through the points is drawn; and if more than three, lines connect the points. (In the future, a spline will be fit to the points if they number greater than three.) Parameters are available to set *color, weight,* and *style*.

text The *text* primitive draws characters. It requires a single point which locates the center of the first character to be drawn. Parameters are *color*, *font*, *textsize*, and *textangle*.

hardware The *hardware* primitive draws hardware characters or gives control commands to a hardware device. A single point locates the beginning location of the *hardware* string.

comment A *comment* is an integer string that is included in a GPS file but causes nothing to be displayed. All GPS files begin with a comment of zero length.

GPS PARAMETERS

color *Color* is an integer value set for *arc, lines,* and *text* primitives.

weight *Weight* is an integer value set for *arc* and *lines* primitives to indicate line thickness. The value **0** is narrow weight, **1** is bold, and **2** is medium weight.

style *Style* is an integer value set for *lines* and *arc* primitives to give one of the five different line styles that can be drawn on TEKTRONIX 4010 series storage tubes. They are:

- **0** solid
- **1** dotted
- **2** dot dashed
- **3** dashed
- **4** long dashed

font An integer value set for *text* primitives to designate the text font to be used in drawing a character string. (Currently *font* is expressed as a four-bit *weight* value followed by a four-bit *style* value.)

textsize *Textsize* is an integer value used in *text* primitives to express the size of the characters to be drawn. *Textsize* represents the height of characters in absolute *universe-units* and is stored at one-fifth this value in the size-orientation (*so*) word (see below).

textangle *Textangle* is a signed integer value used in *text* primitives to express rotation of the character string around the beginning point. *Textangle* is expressed in degrees from the positive x-axis and can be a positive or negative value. It is stored in the size-orientation (*so*) word as a value 256/360 of it's absolute value.

ORGANIZATION

GPS primitives are organized internally as follows:

lines	*cw points sw*
arc	*cw points sw*
text	*cw point sw so* [*string*]
hardware	*cw point* [*string*]
comment	*cw* [*string*]

cw *Cw* is the control word and begins all primitives. It consists of four bits that contain a primitive-type code and twelve bits that contain the word-count for that primitive.

point(s) *Point(s)* is one or more pairs of integer coordinates. *Text* and *hardware* primitives only require a single *point*. *Point(s)* are values within a Cartesian plane or *universe* having 64K (-32K to +32K) points on each axis.

sw *Sw* is the style-word and is used in *lines, arc,* and *text* primitives. For all three, eight bits contain *color* information. In *arc* and *lines* eight bits are divided as four bits *weight* and four bits *style*. In the *text* primitive eight bits of *sw* contain the *font*.

so *So* is the size-orientation word used in *text* primitives. Eight bits contain text size and eight bits contain text rotation.

string *String* is a null-terminated character string. If the string does not end on a word boundary, an additional null is added to the GPS file to insure word-boundary alignment.

SEE ALSO

graphics(1G), stat(1G), toc(1G) in the *User's Reference Manual*.

[This page 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

passwd(4).
passwd(1) in the *User's Reference Manual.*
newgrp(1M) in the *System Administrator's Reference Manual.*

[This page 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 *gettys* 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

exec(2), open(2), signal(2).
getty(1M), init(1M) in the *System Administrator's Reference Manual*.
sh(1), who(1) in the *User's Reference Manual*.

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 could have 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 last file stat
                                 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 left blank.]

NAME

USER-DEFD, USER-DEFD_dict, USER-DEFD_fold

DESCRIPTION

Isort(1) (international sort) uses sorting sequences defined in language tables. If the -IU option is specified, isort uses the tables in the files listed above.

BUILDING ISORT TABLES

The following three tables are found in /usr/spool/isort/TABLES:

USER-DEFD
USER-DEFD_dict (used by isort with -d option)
USER-DEFD_fold (used by isort with -f option)

The predefined tables for the languages listed in isort(1) are also found in /usr/spool/isort/TABLES. You may find them useful references for building your user-defined tables.

Following is an excerpt from the table USER-DEFD:

```
/*    0101  */  0101  0        /* 'A' */
/*    0102  */  0102  0        /* 'B' */
/*    0103  */  0103  0        /* 'C' */
/*    0104  */  0104  0        /* 'D' */
/*    0105  */  0105  0        /* 'E' */
```

The "/*" and "*/" must be in the first and third positions respectively or isort will abort and return an illegal language file error.

The first column of numbers in each table is the ASCII equivalent in octal notation, and must not be changed. The second and third columns of numbers can be changed if necessary to establish a user-defined collating sequence. Normally, entries in the third column will be zero, but in certain languages diacritical marks will require that you enter a different number in the third column. For example, an a with an umlaut is sorted alphabetically as ae. Therefore, the octal numbers for both a and e must be entered; the number for a will be in the second column, and the number for e will replace the zero in the third column. The column of ASCII characters identifies the letter, digit, or symbol that the octal notation signifies.

The table in USER-DEFD establishes a collating sequence for all letters, numbers, and symbols, and distinguishes between uppercase and lowercase letters. The table in USER-DEFD_dict treats only letters, digits, and spaces, by setting everything else to zero; it distinguishes between uppercase and lowercase. The table in USER-DEFD_fold treats uppercase and lowercase letters as identical by setting them to the same value.

The following rules must be observed as you edit the files to establish the desired collation order:

1) Octal 00 to 015 and octal 040 are reserved; no entry in column 2 and 3 can be associated with these values.

2) The digits (0-9) should not be changed from their assigned value of 060-071.

3) Uppercase and lowercase alphabetic characters should be 040 (octal) apart to facilitate building tables to fold lowercase to uppercase.

SEE ALSO

isort(1).

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) in the *User's Reference Manual*.

[This page 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 following example 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 args 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 25          /* max # of processes per user */
#define CLK_TCK   100         /* # of clock ticks per second */
#define DBL_DIG   16          /* digits of precision of 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 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  32767       /* max # of links to a 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 chars in a file name */
#define OPEN_MAX  20          /* max # of files open */
#define PASS_MAX  8           /* max # of chars in a password */
#define PATH_MAX  256         /* max # of chars in a path name */
#define PID_MAX   30000       /* 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  -32767     /* min decimal value of a short */
#define STD_BLK   1024       /* # bytes in a physical
                                I/O block */
#define SYS_NMLN  9          /* # of chars in uname-returned
                                strings */
#define UID_MAX   30000      /* max value for a user or
                                group ID */
#define USI_MAX   4294967296 /* max decimal value of an
                                unsigned */
#define WORD_BIT  32         /* # of bits in a word or int */
```

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)]. An example 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

master - master configuration database

DESCRIPTION

The *master* configuration database is a collection of files. Each file contains configuration information for a device or module that may be included in the system. A file is named with the module name to which it applies. This collection of files is maintained in a directory called **/etc/master.d**. Each individual file has an identical format. For convenience, this collection of files will be referred to as the *master* file, as though it was a single file. This will allow a reference to the *master* file to be understood to mean the *individual file* in the **master.d** directory that corresponds to the name of a device or module. The file is used by the *mkboot*(1M) program to obtain device information to generate the device driver and configurable module files. It is also used by the *sysdef*(1M) program to obtain the names of supported devices. *master* consists of two parts; they are separated by a line with a dollar sign ($) in column 1. Part 1 contains device information for both hardware and software devices, and loadable modules. Part 2 contains parameter declarations used in part 1. Any line with an asterisk (*) in column 1 is treated as a comment.

Part 1, Description

Hardware devices, software drivers and loadable modules can be defined with a line containing information found in the example below. Field 1 must begin in the left most position on the line. Fields are separated by white space (tab or blank). An example of a typical format for these files is shown here.

Field 1: element characteristics:

o	specify only once
r	required device
b	block device
c	character device
a	generate segment descriptor array
t	initialize cdevsw[].d_ttys
s	software driver
f	STREAMS driver

	m	STREAMS module
	x	not a driver; a loadable module
	number	The first interrupt vector for an integral device
Field 2:	number of interrupt vectors required by a hardware device: "-" if none.	
Field 3:	handler prefix (4 chars. maximum)	
Field 4:	software driver external major number; "-" if not a software driver, or to be assigned during execution of *drvinstall*(1M)	
Field 5:	number of sub-devices per device; "-" if none	
Field 6:	interrupt priority level of the device; "-" if none	
Field 7:	dependency list (optional); this is a comma separated list of other drivers or modules that must be present in the configuration if this module is to be included	

For each module, two classes of information are required by *mkboot*(1M): external routine references and variable definitions. Routine and variable definition lines begin with white space and immediately follow the initial module specification line. These lines are free form, thus they may be continued arbitrarily between non-blank tokens as long as the first character of a line is white space.

Part 1, Routine Reference Lines

If the UNIX system kernel or other dependent module contains external references to a module, but the module is not configured, then these external references would be undefined. Therefore, the *routine reference* lines are used to provide the information necessary to generate appropriate dummy functions at boot time when the driver is not loaded.

Example *Routine references* are defined as follows:

Field 1: routine name ()

Field 2: the routine type: one of

```
{}  routine_name(){}
{nosys}
    routine_name(){return nosys();}
{nodev}
    routine_name(){return nodev();}
```

```
{false}
    routine_name(){return 0;}
{true}
    routine_name(){return 1;}
```

Part 1, Variable Definition Lines

Variable definition lines are used to generate all variables required by the module. The variable generated may be of arbitrary size, be initialized or not, or be arrays containing an arbitrary number of elements.

Example *variable references* are defined as follows:

Field 1:	variable_name
Field 2:	[expr] - optional field used to indicate array size
Field 3:	(length) - required field indicating the size of the variable
Field 4:	={ expr,... } - optional field used to initialize individual elements of a variable

The *length* field is mandatory. It is an arbitrary sequence of length specifiers, each of which may be one of the following:

%i	an integer
%l	a long integer
%s	a short integer
%c	a single character
%number	a field which is *number* bytes long
%number c	a character string which is *number* bytes long

For example, the length field

```
( %8c %l %0x58 %l %c %c )
```

could be used to identify a variable consisting of a character string 8-bytes long, a long integer, a 0x58 byte structure of any type, another long integer, and two characters. Appropriate alignment of each % specification is performed (%number is word aligned) and the variable length is rounded up to the next word boundary during processing.

The expressions for the optional array size and initialization are infix expressions consisting of the usual operators for addition, subtraction, multiplication, and division: +, -, *, and /. Multiplication and division have the higher precedence, but

parentheses may be used to override the default order. The builtin functions *min* and *max* accept a pair of expressions, and return the appropriate value. The operands of the expression may be any mixture of the following:

&name	address of name where *name* is any symbol defined by the kernel, any module loaded or any variable definition line of any module loaded
#name	sizeof name where *name* is any variable name defined by a variable definition for any module loaded; the size is that of the individual variable--not the size of an entire array
#C	number of controllers present; this number is determined by the EDT for hardware devices, or by the number provided in the system file for non-hardware drivers or modules
#C(name)	number of controllers present for the module *name*; this number is determined by the EDT for hardware devices, or by the number provided in the system file for non-hardware drivers or modules
#D	number of devices per controller taken directly from the current master file entry
#D(name)	number of devices per controller taken directly from the master file entry for the module *name*
#M	the internal major number assigned to the current module if it is a device driver; zero of this module is not a device driver
#M(name)	the internal major number assigned to the module *name* if it is a device driver: zero if that module is not a device driver
name	value of a parameter as defined in the second part of *master*
number	arbitrary number (octal, decimal, or hex allowed)
string	a character string enclosed within double quotes (all of the character string conventions supported by the C language are allowed); this operand has a value which is the address of a character array containing

the specified string

When initializing a variable, one initialization expression should be provided for each %i, %l, %s, or %c of the length field. The only initializers allowed for a '%number c' are either a character string (the string may not be longer than *number*), or an explicit zero. Initialization expressions must be separated by commas, and variable initialization will proceed element by element. Note that %number specification cannot be initialized--they are set to zero. Only the first element of an array can be initialized, the other elements are set to zero. If there are more initializers than size specifications, it is an error and execution of the *mkboot*(1M) program will be aborted. If there are fewer initializations than size specifications, zeros will be used to pad the variable. For example:

```
={ "V2.L1", #C*#D, max(10,#D), #C(OTHER), #M(OTHER)
}
```

would be a possible initialization of the variable whose length field was given in the preceding example.

Part 2, Description

Parameter declarations may be used to define a value symbolically. Values can be associated with identifiers and these identifiers may be used in the *variable definition* lines.

Parameters are defined as follows:

Field 1:	identifier (8 characters maximum)
Field 2:	=
Field 3:	value, the value may be a number (decimal, octal, or hex allowed), or a string

EXAMPLE

A sample *master* file for a tty device driver would be named "**atty**" if the device appeared in the EDT as "**ATTY**". The driver is a character device, the driver prefix is **at**, two interrupt vectors are used, and the interrupt priority is 6. In addition, another driver named "*ATLOG*" is necessary for the correct operation of the software associated with this device.

```
*FLAG #VEC PREFIX SOFT #DEV IPL DEPENDENCIES/VARIABLES
 tca     2    at    -    2   6  ATLOG
                                 atpoint(){false}
                                 at_tty[#C*#D] (%0x58)
                                 at_cnt(%i) ={ #C*#D }
                                 at_logmaj(%i) ={ #M(ATLOG) }
                                 at_id(%8c) ={ ATID }
                                 at_table(%i%1%31%s)
                                     ={ max(#C,ATMAX),
                                        &at_tty,
                                        #C }
$
ATID = "fred"
ATMAX = 6
```

This *master* file will cause a routine named *atpoint* to be generated by the boot program if the **ATTY** driver is not loaded, and there is a reference to this routine from any other module loaded. When the driver is loaded, the variables *at_tty*, *at_cnt*, *at_logmaj*, *at_id*, and *at_table* will be allocated and initialized as specified. Due to the **t** flag, the *d_ttys* field in the character device switch table will be initialized to point to **at_tty** (the first variable definition line contains the variable whose address will be stored in *d_ttys*). The ATTY driver would reference these variables by coding:

```
extern struct tty at_tty[];
extern int at_cnt;
extern int at_logmaj;
extern char at_id[8];
extern struct {
        int member1;
        struct tty *member2;
        char junk[31];
        short member3;
        } at_table;
```

FILES

/etc/master.d/*

SEE ALSO

system(4).

mkboot(1M), sysdef(1M) in the *System Administrator's Reference Manual.*

[This page left blank.]

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) in the *System Administrator's Reference Manual*.

[This page 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.

FILES

/etc/passwd

SEE ALSO

a64l(3C), getpwent(3C), group(4).
login(1), passwd(1) in the *User's Reference Manual.*

NAME

plot - graphics interface

DESCRIPTION

Files of this format are produced by routines described in *plot*(3X) and are interpreted for various devices by commands described in *tplot*(1G). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by four bytes representing the **x** and **y** values; each value is a signed integer. The last designated point in an **l, m, n,** or **p** instruction becomes the "current point" for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in *plot*(3X).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes [see *tplot*(1G)].

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a new-line.

e erase: Start another frame of output.

f linemod: Take the following string, up to a new-line, as the style for drawing further lines. The styles are "dotted", "solid", "longdashed", "shortdashed", and "dotdashed". Effective only for the **-T4014** and **-Tver** options of *tplot*(1G) (TEKTRONIX 4014 terminal and Versatec plotter).

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *tplot*(1G). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points

outside may be displayable on devices whose face is not square.

DASI 300	space(0, 0, 4096, 4096);
DASI 300s	space(0, 0, 4096, 4096);
DASI 450	space(0, 0, 4096, 4096);
TEKTRONIX 4014	space(0, 0, 3120, 3120);
Versatec plotter	space(0, 0, 2048, 2048);

SEE ALSO

plot(3X), gps(4), term(5).
graph(1G), tplot(1G) in the *User's Reference Manual.*

WARNING

The plotting library *plot*(3X) and the curses library *curses*(3X) both use the names erase() and move(). The curses versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3X) code, and/or #undef move() and erase() in the *plot*(3X) code.

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

terminfo(4), timezone(4), environ(5), term(5).
env(1), login(1), mail(1), sh(1), stty(1), su(1), tput(1) in the *User's Reference Manual.*
su(1M) in the *System Administrator's Reference Manual.*
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

/usr/spool/lp/prsetup

DESCRIPTION

Ilp(1) (international printer spooler) uses the international character set associated with a specific printer. The association between a printer and its character set is specified in /usr/spool/lp/prsetup.

INSERTING ENTRIES IN PRSETUP

Each entry in *prsetup* is a line with three fields, separated by spaces or tabs. The first field is the logical printer name (e.g., pr1). The second field is the printer model number (e.g., 031 for a Model 31 printer). The third field is the international character set (e.g., SPANISH).

Character Sets Supported

The following European character sets are supported:

BRITISH	ITALIAN
CANADIAN	NORTH
DANISH	NORWEGIAN
DUTCH	SPANISH
FINNISH	SWEDISH
FLEMISH	SWISS (FRENCH)
FRENCH-BELGIAN	SWISS (GERMAN)
GERMAN	

Models

Supported entries for printer model are noted in the "printer model" column below:

Printer Model	*Menu/lpadmin Model Name*
25B	ssp25 or spp25
25C	ssp25 or spp25
031	ssp31
035	ssp35 or spp35
037	ssp37 or spp37
047	ssp47 or ssp47hs
105	ssp105 or spp105
115	ssp115 or spp115
789	ssp789

EXAMPLE

To associate the Spanish character set with a Model 31 printer known logically as pr1, place the following line in /usr/spool/lp/prsetup:

```
pr1 031 SPANISH
```

SEE ALSO

ilp(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. The following is an example of the possible format if relocation information is present.

```
struct    reloc
{
          long      r_vaddr ;    /* (virtual) address of ref */
          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

rfmaster - Remote File Sharing name server master file

DESCRIPTION

The **rfmaster** file is an ASCII file that identifies the hosts that are responsible for providing primary and secondary domain name service for Remote File Sharing domains. This file contains a series of records, each terminated by a newline; a record may be extended over more than one line by escaping the newline character with a backslash ("\"). The fields in each record are separated by one or more tabs or spaces. Each record has three fields:

name type data

The type field, which defines the meaning of the *name* and *data* fields, has three possible values:

p The **p** type defines the primary domain name server. For this type, *name* is the domain name and *data* is the full host name of the machine that is the primary name server. The full host name is specified as *domain.nodename*. There can be only one primary name server per domain.

s The **s** type defines a secondary name server for a domain. *Name* and *data* are the same as for the **p** type. The order of the **s** entries in the **rfmaster** file determines the order in which secondary name servers take over when the current domain name server fails.

a The **a** type defines a network address for a machine. *Name* is the full domain name for the machine and *data* is the network address of the machine. The network address can be in plain ASCII text or it can be preceded by a \x to be interpreted as hexadecimal notation. (See the documentation for the particular network you are using to determine the network addresses you need.)

There are at least two lines in the **rfmaster** file per domain name server: one **p** and one **a** line, to define the primary and its network address. There should also be at least one secondary name server in each domain.

This file is created and maintained on the primary domain name server. When a machine other than the primary tries to

start Remote File Sharing, this file is read to determine the address of the primary. If **rfmaster** is missing, the **-p** option of **rfstart** must be used to identify the primary. After that, a copy of the primary's **rfmaster** file is automatically placed on the machine.

Domains not served by the primary can also be listed in the **rfmaster** file. By adding primary, secondary, and address information for other domains on a network, machines served by the primary will be able to share resources with machines in other domains.

A primary name server may be a primary for more than one domain. However, the secondaries must then also be the same for each domain served by the primary.

Example

An example of an **rfmaster** file is shown below. (The network address examples, *comp1.serve* and *comp2.serve*, are STAR-LAN network addresses.)

```
ccs          p     ccs.comp1
ccs          s     ccs.comp2
ccs.comp2    a     comp2.serve
ccs.comp1    a     comp1.serve
```

NOTE: If a line in the **rfmaster** file begins with a # character, the entire line will be treated as a comment.

FILES

/usr/nserve/rfmaster

SEE ALSO

rfstart(1M) in the *System Administrator's Reference Manual.*

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
<pgmr> **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 newlines. 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. An example 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 reloca-
                                       tion */
        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).

[This page 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. An example C structure is given below.

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

struct  syment
{
```

```
    union                           /* 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

system - system configuration information table

DESCRIPTION

This file is used by the **boot** program to obtain configuration information that cannot be obtained from the equipped device table (EDT) at system boot time. This file generally contains a list of software drivers to include in the load, the assignment of system devices such as *pipedev* and *swapdev,* as well as instructions for manually overriding the drivers selected by the self-configuring boot process.

The syntax of the system file is given below. The parser for the **/etc/system** file is case sensitive. All upper case strings in the syntax below should be upper case in the **/etc/system** file as well. Nonterminal symbols are enclosed in angle brackets "< >" while optional arguments are enclosed in square brackets "[]". Ellipses "..." indicate optional repetition of the argument for that line.

```
<fname>  ::= pathname

<string> ::= driver file name from /boot or EDT entry name

<device> ::= special device name ¦ DEV(<major>,<minor>)

<major> ::= <number>

<minor> ::= <number>

<number> ::= decimal, octal or hex literal
```

The lines listed below may appear in any order. Blank lines may be inserted at any point. Comment lines must begin with an asterisk. Entries for EXCLUDE and INCLUDE are cumulative. For all other entries, the last line to appear in the file is used -- any earlier entries are ignored.

BOOT: <fname>
specifies the kernel a.out file to be booted.

EXCLUDE: [<string>] ...
specifies drivers to exclude from the load even if the device is found in the EDT.

INCLUDE: [<string> [(<number>)]] ...
specifies software drivers or loadable modules to be included in the load. This is necessary to include the drivers for software "devices". The optional <number> (parenthesis required) specifies the number of "devices" to be controlled by the driver (defaults to 1). This number corresponds to the builtin variable *#c* which may be referred to by expressions in part one of the **/etc/master** file.

ROOTDEV: <device>
identifies the device containing the root file system.

SWAPDEV: <device> <number> <number>
identifies the device to be used as swap space, the block number the swap space starts at, and the number of swap blocks available.

PIPEDEV: <device>
identifies the device to be used for pipe space.

FILES

/etc/system

SEE ALSO

master(4).
crash(1M) in the *System Administrator's Reference Manual.*

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 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377   $  \0
0000540 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377   *  \0
0000560 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0001160 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377   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
```

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

curses(3X), terminfo(4), term(5).
infocmp(1M) in the *Administrator's Reference Manual*.
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	Term-cap 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**/**vl**)
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=$<16*>,
is2=\EU\Ef\E7\E5\E8\El\ENH\EK\E\0\Eo&\0\Eo\47\E,
kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
prot=\El, rep=\Er%p1%c%p2%' '%+%c$<.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,
```

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 | 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:
              %? c1 %t b1 %e c2 %t b2 %e c3 %t b3 %e c4 %t b4 %e b5%;
              ci are conditions, bi 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

curses(3X), printf(3S), term(5).
captoinfo(1M), infocmp(1M), tic(1M), tty(7) in the *System Administrator's Reference Manual*.
tput(1) in the *User's Reference Manual*.
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.

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.

EXAMPLES

/etc/TIMEZONE for the east coast:

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

SEE ALSO

ctime(3C), profile(4).
rc2(1M) in the *System Administrator's Reference Manual.*

[This page 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 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" filed */
/*  when accounting for something other than a process */
/*  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

getut(3C).
login(1), who(1), write(1) in the *User's Reference Manual*.

NAME

intro - introduction to miscellany

DESCRIPTION

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

[This page left blank.]

NAME

ascii - map of ASCII character set

DESCRIPTION

ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, to be printed as needed. It contains (octal is shown first followed by hexidecimal):

```
|000 nul  |001 soh  |002 stx  |003 etx  |004 eot  |005 enq  |006 ack  |007 bel  |
|010 bs   |011 ht   |012 nl   |013 vt   |014 np   |015 cr   |016 so   |017 si   |
|020 dle  |021 dc1  |022 dc2  |023 dc3  |024 dc4  |025 nak  |026 syn  |027 etb  |
|030 can  |031 em   |032 sub  |033 esc  |034 fs   |035 gs   |036 rs   |037 us   |
|040 sp   |041 !    |042 "    |043 #    |044 $    |045 %    |046 &    |047 '    |
|050 (    |051 )    |052 *    |053 +    |054 ,    |055 -    |056 .    |057 /    |
|060 0    |061 1    |062 2    |063 3    |064 4    |065 5    |066 6    |067 7    |
|070 8    |071 9    |072 :    |073 ;    |074 <    |075 =    |076 >    |077 ?    |
|100 @    |101 A    |102 B    |103 C    |104 D    |105 E    |106 F    |107 G    |
|110 H    |111 I    |112 J    |113 K    |114 L    |115 M    |116 N    |117 O    |
|120 P    |121 Q    |122 R    |123 S    |124 T    |125 U    |126 V    |127 W    |
|130 X    |131 Y    |132 Z    |133 [    |134 \    |135 ]    |136 ^    |137 _    |
|140 `    |141 a    |142 b    |143 c    |144 d    |145 e    |146 f    |147 g    |
|150 h    |151 i    |152 j    |153 k    |154 l    |155 m    |156 n    |157 o    |
|160 p    |161 q    |162 r    |163 s    |164 t    |165 u    |166 v    |167 w    |
|170 x    |171 y    |172 z    |173 {    |174 |    |175 }    |176 ~    |177 del  |

|  00 nul |  01 soh |  02 stx |  03 etx |  04 eot |  05 enq |  06 ack |  07 bel |
|  08 bs  |  09 ht  |  0a nl  |  0b vt  |  0c np  |  0d cr  |  0e so  |  0f si  |
|  10 dle |  11 dc1 |  12 dc2 |  13 dc3 |  14 dc4 |  15 nak |  16 syn |  17 etb |
|  18 can |  19 em  |  1a sub |  1b esc |  1c fs  |  1d gs  |  1e rs  |  1f us  |
|  20 sp  |  21 !   |  22 "   |  23 #   |  24 $   |  25 %   |  26 &   |  27 '   |
|  28 (   |  29 )   |  2a *   |  2b +   |  2c ,   |  2d -   |  2e .   |  2f /   |
|  30 0   |  31 1   |  32 2   |  33 3   |  34 4   |  35 5   |  36 6   |  37 7   |
|  38 8   |  39 9   |  3a :   |  3b ;   |  3c <   |  3d =   |  3e >   |  3f ?   |
|  40 @   |  41 A   |  42 B   |  43 C   |  44 D   |  45 E   |  46 F   |  47 G   |
|  48 H   |  49 I   |  4a J   |  4b K   |  4c L   |  4d M   |  4e N   |  4f O   |
|  50 P   |  51 Q   |  52 R   |  53 S   |  54 T   |  55 U   |  56 V   |  57 W   |
|  58 X   |  59 Y   |  5a Z   |  5b [   |  5c \   |  5d ]   |  5e ^   |  5f _   |
|  60 `   |  61 a   |  62 b   |  63 c   |  64 d   |  65 e   |  66 f   |  67 g   |
|  68 h   |  69 i   |  6a j   |  6b k   |  6c l   |  6d m   |  6e n   |  6f o   |
|  70 p   |  71 q   |  72 r   |  73 s   |  74 t   |  75 u   |  76 v   |  77 w   |
|  78 x   |  79 y   |  7a z   |  7b {   |  7c |   |  7d }   |  7e ~   |  7f del |
```

[This page left blank.]

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:

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**.

HOME
Name of the user's login directory, set by *login*(1) from the password file *passwd*(4).

TERM
The kind of terminal for which output is to be prepared. This information is used by commands, such as *mm*(1) or *tplot*(1G), which may exploit special capabilities of that terminal.

TZ Time zone information. The format is **xxx***n***zzz** where **xxx** is standard local time zone abbreviation, *n* is the difference in hours from GMT, and **zzz** is the abbreviation for the daylight-saving local time zone, if any; for example, **EST5EDT**.

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 ALSO

exec(2).
env(1), login(1), sh(1), nice(1), nohup(1), time(1), tplot(1G) in the *User's Reference Manual*.
mm(1) in the *DOCUMENTER'S WORKBENCH Reference Manual*.

[This page left blank.]

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 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 (3rd 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       /* Ck 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

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 3M), as well as various functions in the C Library (Section 3C) 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	$\pi/2$.
M_PI_4	$\pi/4$.
M_1_PI	$1/\pi$.
M_2_PI	$2/\pi$.
M_2_SQRTPI	$2/\pi$.
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(3), matherr(3M), values(5).

[This page left blank.]

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 nonzero 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) in the *User's Reference Manual.*

[This page left blank.]

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_ISVTX  01000   /* save swapped text even after use */
#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. 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. 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

-np	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 (for a complete list, type: **ls -C /usr/lib/terminfo/?**):

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)
SVT1210, UVT1210	UNISYS Video Terminal Model 1210
SVT1220, UVT1220	UNISYS Video Terminal Model 1220
UVT1224	UNISYS Video Terminal Model 1224
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

curses(3X), profile(4), terminfo(4), environ(5).
man(1), sh(1), stty(1), tabs(1), tput(1), tplot(1G), vi(1) in the *User's Reference Manual.*
infocmp(1M) in the *System Administrator's Reference Manual*.
Chapter 10 of the *Programmer's Guide.*

NOTES

Not all programs follow the above naming conventions.

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 of these types 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[10];
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 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 (in most implementations, 0x8000).

HIBITL The value of a long integer with only the high-order bit set (in most implementations, 0x80000000).

HIBITI The value of a regular integer with only the high-order bit set (usually the same as HIBITS or HIBITL).

MAXSHORT The maximum value of a signed short integer (in most implementations, 0x7FFF, decimal 32767).

MAXLONG The maximum value of a signed long integer (in most implementations, 0x7FFFFFFF, decimal 2147483647).

MAXINT The maximum value of a signed regular integer (usually the same as MAXSHORT or 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 MAXARGS     100

/* 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.