NAME

intro – file format description

DESCRIPTION

This section provides a description of the various file formats and macros required by the system. Where possible, the appropriate header file is included with the page. If not included with the page, most of these can be found in the **/usr/include** or **/usr/include/sys** directories.

NAME

L-devices – auto-dialer device table

DESCRIPTION

The file contains a list of devices which may be used to connect to a remote unix. It consists of one line per device, formatted as

name lnxx yyyy speed

where *name* is the device name as used in table *L.sys*(5) (i.e. **ACU**, **switch**, ...), *lnxx* is the line number as specified in the *inittab*(5) file, *yyyy* represents the device number of auto-dialer as specified in **/dev** (i.e. **dn17**, ...), and *speed* is the data link speed (300, 1200, etc).

FILES

/usr/lib/uucp/L-devices

SEE ALSO

cu(1C), uucp(1C), conns(3C), dn(4), L.sys(5)

NOTE: The maximum # of entrys for L-devices is hard coded in the conns() subroutine as MAXDEV. Current limit is 20.

NAME

L-dialcodes − uucp system dialcodes

DESCRIPTION

This table contains a list of prefix dial codes to be used by *uucp*(1C) and *cu*(1C) in dialing the remote *unix*. It consists of one line per each prefix code formatted as follows:

prefix-code digit-string

where *prefix-code* is the name referenced in table *L.sys*(5) and *digit-string* is an arbitrary length string of digits. The letter **w** should be interspersed in the string where dial tone should be received before dialing continues.

FILES

/usr/lib/uucp/L-dialcodes

SEE ALSO

cu(1C), uucp(1C), conns(3C), L.sys(5)

NAME

L.sys − table of connecting uucp systems

DESCRIPTION

This file is used by the *uucp*(1C) and *cu*(1C) commands to establish a connection to a remote UNIX. It contains fields of data for each remote system indicating when a call can be made, which auto dialers may be used (as specified in file *L-devices*), what speed to use, a possible prefix plus digits to be used in conjunction with the *L-dialcodes* table, and the login sequence to be used after carrier is received. For example, the entry:

cbosg Any ACU 1200 cb4712 *LOGIN nuucp password: yyy*

would direct *uucp*, in response to the system name *cbosg*, to try to make the call at *Any* time, to use one of the available *ACU*'s that that is capable of making a *1200* baud call (defined in *L-devices*), to interpret *cb* as a prefix code by reference to the file *L-dialcodes*(5), and dial the prefix and the specified number to make the call. When the prompt *LOGIN* is received from the remote system, *uucp* will respond with the login id *nuucp*; when the prompt *password:* is received, *uucp* will respond with *yyy*; and so forth for each pair of prompts and responses. *Uucp*(1C) can now look in the *L.sys* file for several entries to the same system. Starting with the first entry for a system, *uucp* will try all entries until successful or all entries have failed.

Since this file contains clear (non-encrypted) passwords, the proper read only permission should be assigned to prevent password disclosure.

FILES

/usr/lib/uucp/L.sys

SEE ALSO

/usr/src/cmd/uucp/UUCP_IMP_DESC
/usr/src/cmd/uucp/NETWORK_DESC
cu(1C), uucp(1C), conns(3C), L-devices(5), L-dialcodes(5)

NOTE

The login procedure has been enhanced to permit special characters (i.e. '#'). It may take a *UUCP* guru to decipher some of the newer features.

NAME

a.out − assembler and link editor output

DESCRIPTION

A.out is the output file of the assembler *as* and the link editor *ld*. Both programs make **a.out** executable if there were no errors and no unresolved external references.

This file has four sections: a header, the program and data text, a symbol table, and relocation bits (in that order). The last two may be empty if the program was loaded with the −s option of *ld* or if the symbols and relocation have been removed by *strip*.

The structure of the entry as given in the include file is:

```
/*      @(#)a.out.h     3.3             */
struct  exec {          /* a.out header */
        int             a_magic;        /* magic number */
        unsigned        a_text;         /* size of text segment */
        unsigned        a_data;         /* size of initialized data */
        unsigned        a_bss;          /* size of unitialized data */
        unsigned        a_syms;         /* size of symbol table */
        unsigned        a_entry;        /* entry point */
        char                            a_unused;       /* not used */
        char                            a_hitext;       /* text high bits */
        char                            a_flag;         /* relocation info stripped */
        char                            a_stamp;        /* System environment stamp */
};

/* macro to calculate text size of big files */
#define TSIZE(x) x.a_text + ((long)x.a_hitext << 16)

#define     A_MAGIC1    0407        /* normal */
#define     A_MAGIC2    0410        /* read-only text */
#define     A_MAGIC3    0411        /* separated I&D */
#define     A_MAGIC4    0405        /* overlay */
#define     A_MAGIC0    0401                    /* ldp (UNIX/RT) */

/* ***** in invocation of BADMAG macro, argument should not be a function.***/
#define     BADMAG(X) X.a_magic!=A_MAGIC1 && X.a_magic!=A_MAGIC2 && X.a_magic!=A_MAGIC3 &&

struct      nlist {     /* symbol table entry */
        char        n_name[8];      /* symbol name */
        char        n_type;         /* type flag */
        char                        n_loc;                  /* text area location */
        unsigned    n_value;        /* value */
};

                        /* values for type flag */
#define     N_UNDF      0           /* undefined */
#define     N_ABS       01          /* absolute */
#define     N_TEXT      02          /* text symbol */
#define     N_DATA      03          /* data symbol */
#define     N_BSS       04          /* bss symbol */
#define     N_TYPE      037
#define     N_REG       024         /* register name */
#define     N_FN        037         /* file name symbol */
#define     N_EXT       040         /* external bit, or'ed in */
#define     FORMAT      "%06o"      /* to print a value */

                        /* values for loc flag */
#define N_SWSP0 1       /* text switchable space 0 */
#define N_SWSP1 2       /* text switchable space 1 */
#define N_SWSP2 3       /* text switchable space 2 */
#define N_SWSP3 4       /* text switchable space 3 */
```

The sizes of each segment are in bytes but are even. The size of the header is not included in any of the other sizes.

When a file produced by the assembler or loader is loaded into core for execution, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized bss, the latter being initialized to all 0's), and a stack. The text segment begins at 0 in the core image; the header is not loaded. If the magic number (word 0) is 407, it indicates that the text segment is not to be write-protected and shared, so the data segment is immediately contiguous with the text segment. If the magic number is 410, the data segment begins at the first 0 mod 8K byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same file, they will share the text segment. If the magic number is 411, the text segment is again pure, write-protected, and shared, and moreover instruction and data space are separated; the text and data segment both begin at location 0. See the 11/70 handbook for restrictions which apply to this situation. The magic number 405 indicates an overlay file. On execution, the current processes' text segment is replaced with the text segment from this module.

The stack will occupy the highest possible locations in the core image: from 177776(8) and growing downward. The stack is automatically extended as required. The data segment is only extended as requested by the *break*(2) system call.

The start of the text segment in the file is 20(8); the start of the data segment is $20+S_t$ (the size of the text) the start of the relocation information is $20+S_t+S_d$; the start of the symbol table is $20+2(S_t+S_d)$ if the relocation information is present, $20+S_t+S_d$ if not.

The symbol table consists of 6-word entries. The first four words contain the ASCII name of the symbol, null-padded(n_name). The next byte is a flag indicating the type of symbol(n_type).

The next byte is a flag indicating the switchable text location for **UNIX** with switchable text areas.

The last word of a symbol table entry contains the value of the symbol.

If the symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader *ld* as the name of a common region whose size is indicated by the value of the symbol.

The value of a word in the text or data portions which is not a reference to an undefined external symbol is exactly that value which will appear in core when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation bits for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

If relocation information is present, it amounts to one word per word of program text or initialized data. There is no relocation information if the 'suppress relocation' flag in the header is on.

Bits 3-1 of a relocation word indicate the segment referred to by the text or data word associated with the relocation word:

00	indicates the reference is absolute
02	indicates the reference is to the text segment
04	indicates the reference is to initialized data
06	indicates the reference is to bss (uninitialized data)
10	indicates the reference is to an undefined external symbol.

Bit 0 of the relocation word indicates if *on* that the reference is relative to the pc (e.g. 'clr x'); if *off*, that the reference is to the actual symbol (e.g., 'clr *$x').

The remainder of the relocation word (bits 15-4) contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

The system environment stamp (see *stamp*(1)) determines which of several possible interpretations the operating system will give to system calls from the executing process.

SEE ALSO

as(1), ld(1), nm(1), stamp(1), strip(1)

NAME

acct − accounting file

DESCRIPTION

When a process terminates, an accounting record is written into the accounting files **/usr/adm/acct** if system accounting has been activated. This file may be summarized by using *sa*(1). The format of this file is:

```
struct acct{
char    ac_comm[14]   /* name of command */
char    ac_flag       /* unused */
char    ac_uid        /* real userid */
long    ac_date       /* start time of command */
long    ac_etime      /* elapsed time in seconds */
long    ac_utime      /* user cpu time (1/60 sec) */
long    ac_stime      /* system time (1/60 sec) */
long    ac_dread      /* disk reads */
long    ac_dwrit      /* disk writes */
};
```

SEE ALSO

accton(1), sa(1), acct(2)

NAME

ar — archive file format

DESCRIPTION

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

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number is 0177545(8) (it was chosen to be unlikely to occur anywhere else). The header of each file is 26 bytes long:

```
/*              @(#)ar.h        2.1             */
#define         ARMAG           0177545
struct ar_hdr {
                char            ar_name[14];
                long            ar_date;
                char            ar_uid;
                char            ar_gid;
                int             ar_mode;
                long            ar_size;
};
```

Each file begins on a word boundary; a null byte 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.

SEE ALSO

ar(1), ld(1), strip(1)

FILES

/usr/include/ar.h

NAME

core − format of core image file

DESCRIPTION

UNIX 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).

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 **/usr/include/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. Attached *maus*(2) segments are not dumped.

The format of the information in the first section is described by the *user* structure of the system, defined in **/usr/include/sys/user.h**. The important stuff not detailed therein is the locations of the registers, which are outlined in **/usr/include/sys/reg.h**.

In general, the debugger *adb*(1) is sufficient to deal with core images.

SEE ALSO

adb(1), signal(2)

NAME

cpio − format of cpio archive

DESCRIPTION

The *header* structure is:

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

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

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

SEE ALSO

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

NAME

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

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

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

SEE ALSO

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

NAME

crontab – table of chronological events to be executed.

DESCRIPTION

Crontab is examined by the process *cron* at a specified intervals to determine if the system clock contains the same time as any of the entries in the table. If it does, the file specified in the table entry is executed. The file consists of one line per entry, with each entry consisting of 6 columns to be separated by spaces. The format is as follows:

minutes past hour (0-59)
hour of day (0-23)
day of month (0-31)
month (1-12)
day of week (0=Sunday 1=Monday,etc)
command to be executed

An asterisk can be used in those columns where a "don't care" condition is desired (except for command to be executed).

FILES

/usr/lib/crontab

SEE ALSO

cron(1)

NAME

d_passwd − dial up password file

DESCRIPTION

D_passwd contains the encrypted password to be entered by dialup terminals defined in **/etc/dialups**. The file format is of the form

[/shell: encrypted passowrd:]

Where *shell* is the current shell process and *password* is added by editing in the encrypted password. This password may be obtained by adding the password to some entry in the password file (using the *passwd*(1) command) and then copying the encrypted text into *d_passwd*.

LOGIN will choose the appropriate line from this file depending on the shell assigned to the user identification in the passwd file. The entry for **/bin/sh** must always be present and is used as the default if an entry for a non-standard shell is not present.

FILES

/etc/d_passwd

SEE ALSO

login(1), dialups(5)

NAME

dialups – list of dialup lines

DESCRIPTION

/etc/dialups contains a list of terminals which should be prompted at login for a dialup passwd. The file is formatted as one line per terminal as:

/dev/ln*xx*

where **ln***xx* is the device associated with the terminal.

FILES

/etc/dialups

SEE ALSO

login(1), d_passwd(5)

NAME

dir − format of directories

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*(5)). The structure of a directory entry as given in the include file is:

```
/*              @(#)dir.h       3.1             */
#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(5)

FILES

/usr/include/sys/dir.h

NAME

dump – incremental dump tape format

DESCRIPTION

The *mhdump* and *mhrestor* commands are used to write and read incremental dump magnetic tapes.

The dump tape consists of a header record, some bit mask records, a group of records describing filesystem directories, a group of records describing filesystem files, and some records describing a second bit mask.

The header record and the first record of each description have the format described by the structure included by

#include <dumprestor.h>

This include file has the following contents.

```
/*              @(#)dumprestor.h            2.1         */
#define NTREC                   20
#define MLEN                    16
#define MSIZ    4096

#define TS_TAPE                 1
#define TS_INODE                2
#define TS_BITS                 3
#define TS_ADDR                 4
#define TS_END                  5
#define TS_CLRI                 6
#define MAGIC                   (int)60011
#define CHECKSUM                (int)84446
struct          spcl
{
                int             c_type;
                time_t          c_date;
                time_t          c_ddate;
                int             c_volume;
                daddr_t         c_tapea;
                ino_t           c_inumber;
                int             c_magic;
                int             c_checksum;
                struct          dinode          c_dinode;
                int             c_count;
                char            c_addr[BSIZE];
} spcl;

struct          idates
{
                char            id_name[16];
                char            id_incno;
                time_t          id_ddate;
};
```

NTREC is the number of 512 byte blocks in a physical tape record. *MLEN* is the number of bits in a bit map word. *MSIZ* is the number of bit map words.

The *TS_* entries are used in the *c_type* field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TAPE
Tape volume label

TS_INODE
A file or directory follows. The *c_dinode* field is a copy of the disk inode and contains

bits telling what sort of file this is.

TS_BITS A bit mask follows. This bit mask has a one bit for each inode that was dumped.

TS_ADDR
A subblock to a file *(TS_INODE)*. See the description of *c_count* below.

TS_END End of tape record.

TS_CLRI A bit mask follows. This bit mask contains a one bit for all inodes that were empty on the file system when dumped.

MAGIC All header blocks have this number in *c_magic*.

CHECKSUM
Header blocks checksum to this value.

The fields of the header structure are as follows:

c_type The type of the header.

c_date The date the dump was taken.

c_ddate The date the file system was dumped from.

c_volume The current volume number of the dump.

c_tapea The current block number of this record. This is counting 512 byte blocks.

c_inumber
The number of the inode being dumped if this is of type *TS_INODE*.

c_magic This contains the value *MAGIC* above, truncated as needed.

c_checksum
This contains whatever value is needed to make the block sum to *CHECKSUM*.

c_dinode This is a copy of the inode as it appears on the file system.

c_count This is the count of characters following that describe the file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is non-zero. If the block was not present on the file system no block was dumped and it is replaced as a hole in the file. If there is not sufficient space in this block to describe all of the blocks in a file, *TS_ADDR* blocks will be scattered through the file, each one picking up where the last left off.

c_addr This is the array of characters that is used as described above.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a *TS_END* block and then the tapemark.

The structure *idates* describes an entry of the file where dump history is kept.

SEE ALSO

mhdump(1), mhrestor(1), fs(5)

FILES

/usr/include/sys/types.h, /usr/include/sys/ino.h, /usr/include/dumprestor.h

NAME

errfile – error-log file format

DESCRIPTION

When hardware errors are detected by the system, an error record is generated and passed to the error-logging daemon for recording in the error log for later analysis. The default error log is **/errlog/errfile.**

The format of an error record depends on the type of error that was encountered. Every record, however, has a header with the following format:

```
struct errhdr {
        int             e_type;         /* record type */
        int             e_len;          /* bytes in record (inc hdr) */
        time_t          e_time;         /* time of day */
};
```

The permissible record types are as follows:

```
#define E_GOTS  010             /* Start for UNIX/TS */
#define E_GORT  011             /* Start for UNIX/RT */
#define E_STOP  012             /* Stop */
#define E_TCHG  013             /* Time change */
#define E_CCHG  014             /* Configuration change */
#define E_BLK   020             /* Block device error */
#define E_STRAY 030             /* Stray interrupt */
#define E_PRTY  031             /* Memory parity */
#define E_OVFL  040             /* Software table overflow */
#define E_PRDEV 041             /* File system error */
#defint E_POWER 042             /* Power-fail restart */
```

Some records in the error file are of an administrative nature. These include the startup record that is entered into the file when logging is activated, the stop record that is written if the daemon is terminated "gracefully", and the time-change record that is used to account for changes in the system's time-of-day. These records have the following formats:

```
struct estart {
        struct errhdr   e_hdr;          /* record header */
        int             e_cpu;          /* cpu type */
        int             e_mmr3;         /* contents mem mgmt reg 3 */
        long            e_syssize;      /* 11/70 system memory size */
        int             e_bconf;        /* block dev configuration */
};

struct eend {
        struct errhdr   e_hdr;          /* record header */
        int             e_werr;         /* number of daemon write errors */
};

struct etimchg {
        struct errhdr   e_hdr;          /* record header */
        time_t          e_ntime;        /* new time */
};
```

Stray interrupts cause a record with the following format to be logged in the file:

```
struct estray {
        struct errhdr   e_hdr;          /* record header */
        physadr         e_saddr;        /* stray loc or device addr */
        int             e_sbacty;       /* active block devices */
};
```

Memory subsystem error on 11/70 processors cause the following record to be generated:

```
struct eparity {
        struct errhdr   e_hdr;          /* record header */
        int             e_parreg[4];    /* memory subsys registers */
};
```

Error records for block devices have the following format:

```
struct eblock {
        struct errhdr   e_hdr;          /* record header */
        dev_t           e_dev;          /* "true" major + minor dev no */
        physadr         e_regloc;       /* controller address */
        int             e_bacty;        /* other block I/O activity */
        struct iostat {
          long          io_ops;         /* number read/writes */
          long          io_misc;        /* number "other" operations */
          unsigned      io_unlog;       /* number unlogged errors */
        }               e_stats;
        int             e_bflags;       /* read/write, error, etc */
        int             e_cyloff;       /* logical dev start cyl */
        daddr_t         e_bnum;         /* logical block number */
        unsigned        e_bytes;        /* number bytes to transfer */
        long            e_memadd;       /* buffer memory address */
        unsigned        e_rtry;         /* number retries */
        int             e_nreg;         /* number device registers */
};
```

The following values are used in the flags word:

```
#define E_WRITE 0       /* Write operation */
#define E_READ  1       /* Read operation */
#define E_NOIO  02      /* No I/O pending */
#define E_PHYS  04      /* Physical I/O */
#define E_MAP   010     /* Unibus map in use */
#define E_ERROR 020     /* I/O failed */
```

The "true" major device numbers that identify the failing device are as follows:

```
#define RK0     0
#define RP0     1
#define RF0     2
#define TM0     3
#define TC0     4
#define HP0     5
#define HT0     6
#define HS0     7
#define RLO     8
```

File system soft errors generate records of the following format:

```
struct eprdev {
        struct errhdr   e_hdr;          /* record header */
        short           e_missed;       /* errors not logged since preceding record */
        dev_t           e_fsdev;        /* device with filesystem in error */
        short           e_fserr;        /* type of error */
};
```

Values for e_fserr include:

```
#define E_FSBB  0               /* Bad block */
#define E_FSBC  1               /* Bad count */
#define E_FSNS  2               /* No space */
#define E_FSOI  3               /* Out of inodes */
```

Table overflow errors generate records of the following format:

```
struct eovfl {
        struct errhdr   e_hdr;          /* record header */
        short           e_missed;       /* errors not logged since preceeding record */
        short           e_tabt;         /* type of error */
};
```

Values for e_tabt are:

```
#define E_FILEO  0              /* File table overflow */
#define E_PROCO  1              /* Process table overflow */
#define E_INODEO2               /* Inode table overflow */
#define E_TEXTO  3              /* Text table overflow */
```

Powerfail − restart records have the format:

```
struct e_power {
        struct errhdr   e_hdr;          /* record header */
};
```

SEE ALSO

errdemon(1M)

NAME

filesystem − format of system volume

DESCRIPTION

Every file system storage volume (e.g. RP04 disk) has a common format for certain vital information. Every such volume is divided into a certain number of 256 word (512 byte) blocks. Block 0 is unused and is available to contain a bootstrap program or other information.

Block 1 is the *super block*. Starting from its first word, the format of a super-block is:

```
/*              @(#)filsys.h    3.1             */
/*
 * Definition of the unix super block.
 * The root super block is allocated and
 * read in iinit/alloc.c. Subsequently
 * a super block is allocated and read
 * with each mount (smount/sys3.c) and
 * released with unmount (sumount/sys3.c).
 * A disk block is ripped off for storage.
 * See alloc.c for general alloc/free
 * routines for free list and I list.
 */
struct          filsys
{
                char            *s_isize;       /* size in blocks of I list */
                char            *s_fsize;       /* size in blocks of entire volume */
                int             s_nfree;        /* number of in core free blocks (0-100) */
                int             s_free[100];    /* in core free blocks */
                int             s_ninode;       /* number of in core I nodes (0-100) */
                int             s_inode[100];   /* in core free I nodes */
                char            s_flock;        /* lock during free list manipulation */
                char            s_ilock;        /* lock during I list manipulation */
                char            s_fmod;                 /* super block modified flag */
                char            s_ronly;        /* mounted read-only flag */
                long            s_time;                 /* current date of last update */
                int             pad[40];
                int             s_tfree;        /* Total free, for subsystem examination */
                int             s_tinode;       /* Free inodes, for subsystem examination */
                char            s_fname[6];     /* File system name */
                char            s_fpack[6];     /* File system pack name */
};
```

Isize is the number of blocks devoted to the i-list, which starts just after the super-block, in block 2. *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 *free* array contains, in *free[1]*, ... , *free[nfree−1]*, up to 49 numbers of free blocks. *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 *nfree*, and the new block is *free[nfree]*. If the new block number is 0, there are no blocks left, so give an error. If *nfree* became 0, read in the block named by the new block number, replace *nfree* by its first word, and copy the block numbers in the next 50 longs into the *free* array. To free a block, check if *nfree* is 50; if so, copy *nfree* and the *free* array into it, write it out, and set *nfree* to 0. In any event set *free[nfree]* to the freed block's number and increment *nfree*.

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

Ninode is the number of free i-numbers in the *inode* array. To allocate an i-node: if *ninode* is greater than 0, decrement it and return *inode[ninode]*. If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the *inode* array, then try again. To free an i-node, provided *ninode* is less than 100, place its number into *inode[ninode]* and increment *ninode*. If *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 inode is really free or not is maintained in the inode itself.

Tinode is the total free inodes available in the file system.

Flock and *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 *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.

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

Time is the last time the super-block of the file system was changed, and is a double-precision representation of the number of seconds that have elapsed since 0000 Jan. 1, 1970 (GMT). During a reboot, the *time* of the super-block for the root file system is used to set the system's idea of the time.

Fname is the name of the file system and *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, so 8 of them fit into a block. Therefore, i-node i is located in block $(i + 15) / 8$, and begins $64*((i + 15) \pmod 8)$ bytes from its start. 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 inode and its flags, see *inode*(5).

FILES

/usr/include/sys/filsys.h
/usr/include/sys/stat.h
/usr/include/sys/types.h
/usr/include/sys/param.h

SEE ALSO

inode(1), mkfs(1M), stat(2), stat:o(2), inode(5)

NAME

gettydefs − speed and terminal settings used by getty

DESCRIPTION

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

Each entry in the *gettydefs* file has the following format:

label# inital flags # final flags # input speed # output speed # login message # nextlabel

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* is trying to match the second argument. It is often just the speed, such as '1200', at which the terminal is supposed to run, but it needn't be.

inital flags These flags are the initial *ioctl* 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/ioctl.h** (see *ioctl*(2)) under the commands **TTIOCSETP** and **TTIOCSETO** commands. They are **HUPCL XTABS LCASE ECHO CRMOD RAW ODDP EVENP ANYP NLDELAY TBDELAY CRDELAY VTDELAY BSDELAY ALLDELAY TANDEMO HDPLX NOHUP XCLUDE NOSLEEP TANDEMI** and **STDTTY**. These settings remain in effect until *getty* exec's *login*. For the initial modes the state of three of these flags will be set regardless of what the table says, hence they needn't and shouldn't be included in the **initial flags.** These flags are **RAW** and **HUPCL,** which are set on, and **ECHO,** which is set off.

final flags These flags take the same values as the **initial flags** and are set just prior to *getty* switching to *login*. The **HUPCL** flag is always set on and so shouldn't appear in the **final flags** at all.

input speed This specifies the input speed the terminal will be set at for this entry. It can also set character width and number of stop bits. The words *getty* understands in this field and the **output speed** also come from **/usr/include/sys/ioctl.h.** The legal words are **B0 B50 B75 B110 B134 B150 B200 B300 B600 B1200 B1800 B2400 B4800 B9600 EXTA EXTB ONESTOP TWOSTOP BITS5 BITS6 BITS7** and **BITS8 .**

output speed This specifies the output speed the terminal will be set at for this entry. It can also set character width and number of stop bits.

login message This entire field is printed as the login message. Unlike the above fields where white spaces are ignored (white spaces are ' ', '\t', and '\n'), they are included in the **login message** field.

next label If this entry does not specify the correct speed, indicated by having the user type 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 by 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't find the

specified label. If /etc/**gettydefs** itself is missing, there is one built in entry inside *getty* itself, which will bring up a terminal at 300 baud.

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

FILES

/etc/gettydefs

SEE ALSO

getty(1M), ioctl(2)

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 by the *newgrp*(1) command.

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

FILES

/etc/group

SEE ALSO

newgrp(1), passwd(1), crypt(3C), passwd(5)

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** which initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the *shell*.

The lines 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* convention for comments. (See *sh*(1)) Comments for lines which spawn *gettys* are displayed by the *who* 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 state* in which this entry is to be processed. Run states effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a run state or run states in which it is allowed to exist. The *run states* are represented by a number ranging from 0 through 6. As an example, if the system is in *run state* 1, only those entries having a 1 in the *run state* field will be processed. When *init* is requested to change *run states,* all processes which do not have an entry in the *rstate* field for the target *run state* will be sent the warning signal and allowed a 20 second grace period before being forcibly terminated by a kill signal. The *rstate* field can define multiple *run states* for a process by selecting more than one run state in any combination from 0 through 6. If no *run state* is specified, then the process is assumed to be valid at *all run states 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 states*. Entries which have these characters in the *rstate* field are processed only when the *telinit* process requests them to be run (regardless of the current *run state* of the system). They differ from *run states* in that *init* can never enter *run state a, b* or *c*. Also, a request for the execution of any of these processes does not change the current *run state*. 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 state 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 state will cause *init* to ignore this entry.

once Upon *init*'s entering a run state that matches the entry's *rstate*, start the process, do not wait for its termination and when it dies do not restart the process. If upon entering a new run state the process is still running

from a previous run state 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 state 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 only at *init*'s boot time read of the *inittab* file. *Init* is to start the process, wait for its termination and when it dies not restart the process.

powerfail Execute the process assciated with this entry only when *init* receives a power fail signal (*SIGPWR* (*signal*(2))).

powerwait Execute the process associated with this entry only when *init* receives a power fail siganl (*SIGPWR* (*signal*(2))) 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 (*SIGTRM* (*signal*(2))) and wait 20 seconds before forcibly terminating the process via the kill signal (*SIGKIL* (*signal*(2))). 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 states. This is used only with the *a, b,* or *c* values described in the '*rstate*' field.

initdefault

An entry with this **action** verb is only scanned when *init* initially comes up. *Init* uses this entry, if it exists, to determine which *run state* to enter initially. It does this by taking the highest run level specified in the **rstate** field and using that as its initial state. Two points to note. If the **rstate** field is empty, this in interpretted as '0123456' and so *init* will enter *run state* 6. The second point is that the **initdefault** entry cannot specify that *init* start in the *SINGLE USER* state. Also to be noted is that if *init* doesn't find an **initdefault** entry in **/etc/inittab,** then it will request an initial *run state* from the user at reboot time.

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

FILES

/etc/inittab

SEE ALSO

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

NAME

inode − format of an inode

SYNOPSIS

#include <sys/ino.h>

DESCRIPTION

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

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

/*
 * The inode layout as it appears on the disk.
 * This header file in not used by the system, but by programs like
 * ncheck.
 */
struct          inode
{
                int             i_mode;
                char            i_nlink;        /* directory entries */
                char            i_uid;                          /* owner */
                char            i_gid;                          /* group of owner */
                char            i_size0;        /* most significant of size */
                char            *i_size1;       /* least sig */
                int             i_addr[8];      /* device addresses constituting file */
                int             i_atime[2];     /* last access time */
                int             i_mtime[2];     /* last modification time */
};

/* modes */
#define         IALLOC          0100000                         /* file is used */
#define         IFMT            060000                          /* type of file */
#define                         IFDIR           040000          /* directory */
#define                         IFCHR           020000          /* character special */
#define                         IFBLK           060000          /* block special, 0 is regular */
#define         ILARG           010000                          /* large addressing algorithm */
#define         ISUID           04000                           /* set user id on execution */
#define         ISGID           02000                           /* set group id on execution */
#define         ISVTX           01000                           /* save text, event when not current */
#define         IREAD           0400                            /* read, write, execute permissions */
#define         IWRITE          0200
#define         IEXEC           0100
```

FILES

/usr/include/sys/ino.h

SEE ALSO

stat(2), fs(5).

NAME

issue − issue identification file

DESCRIPTION

/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*(1M) and then written to any terminal spawned or respawned from the *lines* file.

FILES

/etc/issue

SEE ALSO

getty(1M), login(1M)

NAME

lfs − format of Logical File System disk area

DESCRIPTION

As described in *lfs*(3C), The Logical File System (LFS) is a fast-access file system that provides contiguous file storage. The LFS disk area is configured by the *mklfs*(1) command. It is broken into several distinct areas:

- Sector 0 is unused and could be used in the future for any purpose desired. It was left empty to be consistent with standard UNIX file systems.
- Sector 1 contains the LFS header which contains all of the parameters input to *mklfs* plus a "magic" number used for detection of major disk overwrites, the starting sector of the free list (see below), and the last lfn created automatically by the LFS.
- The next area contains a file definition entry for each lfn. Each entry contains some flags for the lfn (e.g. allocated or not), the starting LFS block, the number of LFS blocks in the file, and arbitrary user information about the file. A LFS block is defined to be a contiguous area *blkf* sectors long where *blkf* is the LFS block size specified in the *mklfs*(1) command. Since a file is allocated by the user in terms of sectors, the size in sectors is also stored. If the number of file definition entries fitting in a sector is **nfde**, and **nlfn** is the maximum number of files defined by *mklfs*, then the entries take (nlfn + (nfde − 1))/nfde sectors to store.
- The next area is a freelist (one sector long) containing the starting address and size (both in LFS blocks) of unallocated disk space. It is updated when files are created and deleted. After *mklfs* configures the disk, all "overhead" LFS blocks (e.g., header, file definition entries, and the freelist) have been allocated. The first unallocated area thus starts at the first LFS block after the last overhead block and has a size equal to the remaining space on the disk area.
- A bit map is stored next which records the allocated/unallocated status of each LFS block in the system. It is used primarily for checking the sanity of the LFS in case of system crashes. Since there is one bit per LFS block, the bitmap takes
  ((disksize+7)/8 + 511)/512
  sectors where
  disksize = (ncyl * trkf * secf + (blkf−1))/blkf.
  (See the *mklfs*(1) command for definitions of these quantities).
- The last area of the disk contains the files themselves and comprises the remainder of the disk area. As implied above, this area may be subdivided into any number of files from 1 to *nlfn*. The only limit is that the size of a file cannot exceed 32,767 LFS blocks.

FILES

/usr/include/sys/lfsh.h

SEE ALSO

mklfs(1), lfs(3C)

NAME

No longer used. See *inittab*(5).

NAME

lines − script for the init process

DESCRIPTION

The *lines* 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** which initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

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

id:rstate:action:shellcm: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 and comments may be inserted before or after an entry by using the C comment convention (i.e. /* comment */). (Comments are not included in the 512 character limit.) There are no limits (other than maximum entry size) imposed on the number of entries within the *lines* file. The entry fields are:

id This is one or two characters (other than *xx, !B, RL, OT, NT*) used to uniquely identify an entry. For compactness of syntax, however, when spawning terminal processes these characters must be the name of the line the terminal process is to open (e.g., *aa* causes **/dev/ln***aa* to be the line on which a terminal process is spawned). If a line monitor other than **/etc/getty** is spawned this convention should also be observed in order to produce consistent line accounting.

rstate This defines the *run state* in which this entry is to be processed. Run states effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a run state or run states in which it is allowed to exist. The *run states* are represented by a number ranging from 0 through 6. There is a *run state* 7; however, this is predefined as single-user mode and *init* does not scan the lines file in single-user mode so that a process with run state 7 in the lines file is meaningless. As an example, if the system is in *run state* 1, only those entries having a 1 in the *run state* field will be processed. When *init* is requested to change *run states*, all processes which do not have an entry in the *rstate* field for the target *run state* will be sent the warning signal and allowed a 20 second grace period before being forcibly terminated by a kill signal. The *rstate* field can define multiple *run states* for a process by selecting more than one run state in any combination from 0 through 6. The *run states* must appear as an unbroken string (i.e. '024, '3526, '0125, etc.) in the *rstate* field. If no *run state* is specified (no blanks or tabs may appear in the *rstate* field); then the process is assumed to be valid at *all run states*. There are three other values *a, b,* and *c* which can appear in the *rstate* field even though they are not true *run states*. Entries which have these characters in the *rstate* field are processed only when the *telinit* process requests them to be run (regardless of the current *run state* of the system). They differ from *run states* in that *init* can never enter *run state a, b* or *c*. Also, a request for the execution of any of these processes does not change the current *run 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 *lines* file), and when it dies restart the process. If the process currently exists then do nothing and continue scanning the *lines* file.

wait Upon *init's* entering the run state that matches the entry's *rstate*, start the process and wait for its termination. All subsequent reads of the *lines*

The following number registers are available; they are given default values by .TH:

IN Left margin indent relative to subheads (default is 0.5i).

LL Line length including IN (default is 6.5i).

CAVEAT

In addition to the macros, strings, and number registers mentioned above, there are defined a number of internal macros, strings, and number registers. Except for names predefined by *troff*(1) and number registers **d**, **m**, and **y**, all such internal names are of the form *XA*, where *X* is one of),], and }, and *A* stands for any alphanumeric character.

FILES

/usr/lib/macros/an

SEE ALSO

man(1), nroff(1)

NAME

mountpts – general user mount point table

DESCRIPTION

Mountpts resides in directory /etc and contains a table of directories where the general unprivileged user can legally mount file systems.

Each entry is an ASCII line with the full pathname of a directory on it. Whenever an unprivileged user tries to mount a filesystem, the *mount* command reads this file to see whether they are allowed to mount on the directory they have specified.

If **/etc/mountpts** does not exist, then the general unprivileged user will not be allowed to mount any file systems.

SEE ALSO

mount(1)

NAME

mpxio − multiplexed I/O

DESCRIPTION

Data transfers on *mpx* files (see *mpx* (2)) are multiplexed by imposing a record structure on the I/O stream. Each record represents data from/to a particular channel or a control or status message associated with a particular channel.

The prototypical data record read from an *mpx* file is as follows

```
struct input_record {
        short   index;
        short   count;
        short   ccount;
        char    data[];
};
```

where *index* identifies the channel, and *count* specifies the number of characters in *data*. If *count* is zero, *ccount* gives the size of *data*, and the record is a control or status message. Although *count* or *ccount* might be odd, the operating system aligns records on short (i.e. 16-bit) boundaries by skipping bytes when necessary.

Data written to an *mpx* file must be formatted as an array of record structures defined as follows:

```
struct output_record {
        short   index;
        short   count;
        short   ccount;
        char    *data;
};
```

where the data portion of the record is referred to indirectly and the other cells have the same interpretation as in *input_record*.

The control messages listed below may be read from a multiplexed file descriptor. They are presented as two 16-bit integers: the first number is the message code (defined in *<sys/mx.h>*), the second is an optional parameter meaningful only with M_WATCH.

- M_WATCH a process 'wants to attach' on this channel. The second parameter is the 16-bit user-id of the process that executed the open.
- M_CLOSE the channel is closed. This message is generated when the last file descriptor referencing a channel is closed. The *detach* command (see *mpx* (2) should be used in response to this message.
- M_EOT indicates logical end of file on a channel. If the channel is joined to a typewriter, EOT (control-d) will cause the M_EOT message under the conditions specified in *tty* (4) for end of file. If the channel is attached to a process, M_EOT will be generated whenever the process writes zero bytes on the channel.
- M_UBLK is generated for a channel when the internal queues have drained below a threshold.
- M_SIG is generated instead of a normal asynchronous signal on channels that are joined to typewriters. The parameter is the signal number.

Two other messages may be generated by the kernel. As with other messages, the first 16-bit quantity is the message code.

M_OPEN is generated in conjunction with 'listener' mode (see *mpx*(2)). The uid of the calling process follows the message code as with M_WATCH. This is followed by a null-terminated string which is the name of the file being opened.

M_IOCTL is generated for a channel connected to a process when that process executes the *ioctl (fd, cmd, &vec)* call on the channel file descriptor. The M_IOCTL code is followed by the *cmd* argument given to *ioctl* followed by the contents of the structure *vec*. It is assumed, not needing a better compromise at this time, that the length of *vec* is determined by *sizeof (struct sgttyb)* as declared in <*ioctl.h*>.

Two control messages are understood by the operating system. M_EOT may be sent through an *mpx* file to a channel. It is equivalent to propagating a zero-length record through the channel; i.e. the channel is allowed to drain and the process or device at the other end receives a zero-length transfer before data starts flowing through the channel again. M_IOCTL can also be sent through a channel. The format is identical to that described above.

FILES

/usr/include/sys/ioctl.h
/usr/include/sys/mx.h

SEE ALSO

ioctl(2), mpx(2), tty(4)

NAME

mtab — mounted file system table

DESCRIPTION

Mtab resides in directory /etc and contains a table of devices mounted by the *mount* command. *Umount* removes entries.

Each entry is an ASCII line saying what file system was mounted, where it was mounted, who mounted it, at what time it was mounted, and whether the file system is restricted to read-only access and/or whether set user/group ids will take place if something is executed from the mounted file system.

This table is present only so people can look at it. It does not matter to *mount* if there are duplicated entries nor to *umount* if a name cannot be found.

FILES

/etc/mtab

SEE ALSO

mount(1), umount(1)

NAME

news - USENET network news article

DESCRIPTION

There are two formats of news articles: A and B. A format is the only format that old netnews systems can read or write. Systems running the second netnews can read either format and there are provisions for the second netnews to write in A format. A format looks like this:

Afilename
newsgroups
path
date
title
Body of article

Only second netnews systems can read and write B format. B format contains two extra pieces of information: received date and expiration date. The basic structure of a B format file consists of a series of headers and then the body. A header field is defined as a line with a capital letter in the 1st column and a colon somewhere on the line. Unrecognized header fields are ignored. The following fields are recognized:

Header	Information
From:	Path
To:	Newsgroups
Newsgroups:	Newsgroups
Article-I.D.	Unique Identification (filename for A format)
Subject:	Title
Title:	Title
Posted:	Submittal Date
Received:	Receival Date
Expires:	Expiration Date

The default article skeleton looks like this:

From:
Newsgroups:
Title:
Article-I.D.:
Posted:

Expires:
Received:

Body of article

SEE ALSO

inews(1), sendnews(1), uurec(1), readnews(1)

NAME

~/.newsrc - information file for readnews(1) and newscheck(1)

DESCRIPTION

The .newsrc file contains an optional options line for readnews(1) and newscheck(1) and the list of previously read articles.

The options line starts with the word **options**. Then there are the list of options just as they would be on the command line. For instance:

```
options -s all !fa.sf-lovers !fa.human-nets -r
or
options -c -s general all.general fa.unix-wizards btl.blfp
```

The options line, if present, should be the first line of the file.

Each newsgroup that articles have been read from has a line of the form:

newsgroup: range

The range is a list of the articles read. It is basically a list of no.'s seperated by commas with sequential no.'s collapsed with hyphens. For instance:

```
general: 1-78,80,85-90
fa.info-cpm: 1-7
net.news: 1
```

SEE ALSO

readnews(1), newscheck(1)

NAME

nar − archive (library) file format

SYNOPSIS

#include <nar.h>

DESCRIPTION

The archive command *nar* is used to combine several files into one with printable ASCII format headers. This command is provided for compatibility with UNIX 4.0 commands, and is not compatible with the link-editor *ld*.

A file produced by *nar* has a magic string at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

```
/*      @(#)nar.h3.1*/
#define ARMAG  "!<arch>\n"
#define SARMAG 8

#define ARFMAG "`\n"

struct  ar_hdr {
        char    ar_name[16];
        char    ar_date[12];
        char    ar_uid[6];
        char    ar_gid[6];
        char    ar_mode[8];
        char    ar_size[10];
        char    ar_fmag[2];
};
```

The name is a blank-padded string. The *ar_fmag* field contains ARFMAG to help verify the presence of a header. The other fields are left-adjusted, blank-padded numbers. They are decimal except for *ar_mode*, which is octal. The date is the modification date of the file at the time of its insertion into the archive.

Each file begins on a even (0 mod 2) boundary; a new-line is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

There is no provision for empty areas in an archive file.

The encoding of the header is portable across machines. If an archive contains printable files, the archive itself is printable.

SEE ALSO

arcv(1), nar(1), nm(1)

BUGS

File names lose trailing blanks. Most software dealing with archives takes even an included blank as a name terminator.

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, priority, 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. The priority is also included in this field as *pri=x* where *x* is an integer corresponding to the initial shell priority(pri=0 is the default). 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, **/bin/sh** 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 ID's 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, is not used on the CB-UNIX Release 2.0. (It is used on UNIX/TS to require changing the password after a period of weeks.) 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. If 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

login(1), passwd(1), crypt(3C), getpwent(3C), group(5)

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 *plot*(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 *plot*(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 **−T**ver options of *plot*(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 *plot*(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.

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

SEE ALSO

plot(1G), plot(3X), graph(1G).

NAME

powerfail − commands to be executed following powerfail

DESCRIPTION

Following every powerfail sequence, the file **/etc/powerfail** is read and executed as shell script. Shell commands may be provided in this file to perform any user required functions at that point.

The execution of **/etc/powerfail** is under the control of *init*(1M).

FILES

/etc/powerfail

SEE ALSO

init(1M), signal(2), lines(5)

NAME

printers − defines printer options to /etc/lpd

DESCRIPTION

Printers defines for lpd the name, group, device, type, and speed of all legal printers in the spooling system. The file is similar to the lines file in format (i.e., colon separated fields). The format is thus:

name:group:device[:type][:speed]

Name is the name of the printer which may be from 1 to NAMSIZ characters long (defined in lpss.h). *Group* defines the group to which the printer belongs; it must be identical to the group field in qmap for the queue to which the printer is assigned. *Device* is the location in the file system where the printer lives (i.e., /dev/ln32). *Type* is one of a number of supported printer types. The list currently is *terminet, ds40, ti700 lp11,* and *versatec. Speed* is the baud rate of the printer, if applicable.

FILES

/usr/lpd/.printers

NAME

profile − setting up an environment at login time

DESCRIPTION

If a file named **/etc/profile** exists or if your login directory contains a file named **.profile**, the shell commands in both files will be executed (via the shell's **exec .profile**) before your session begins. These files are handy for setting up both common functions to be executed by all users who log on (via **/etc/profile** commands) and commands to be executed on a individual basis (. *profile* commands). The following example is typical (except for the comments):

```
: 'Make some environment variables global'
export MAIL PATH TERM
: 'Set file creation mask'
umask 22
: 'Tell me when new mail comes in'
MAIL=/usr/mail/myname
: 'Add my /bin directory to the shell search sequence'
PATH=$PATH:$HOME/bin
: 'Set terminal type'
echo "terminal: \c"
read TERM
case $TERM in
        300)        stty cr2 nl0 tabs; tabs;;
        300s)       stty cr2 nl0 tabs; tabs;;
        450)        stty cr2 nl0 tabs; tabs;;
        hp)         stty cr0 nl0 tabs; tabs;;
        745|735)    stty cr1 nl1 −tabs; TERM=745;;
        43)         stty cr1 nl0 −tabs;;
        4014|tek)   stty cr0 nl0 −tabs ff1; TERM=4014; echo "\33;";;
        *)          echo "$TERM unknown";;
esac
```

SEE ALSO

env(1), mail(1), sh(1), stty(1), environ(7)

FILES

/etc/profile
$(HOME)/.profile

NAME

qmap − queue to printers map

DESCRIPTION

Qmap contains information vital to the line printer spooling system. It is a file similar in format to the *lines* file. Each line of the file indicates various data about one queue (in colon separated fields). The format is thus:

name:group:bandef:p1[:p2]

Name is the name of the queue. *Group* is the group name to which the queue is assigned. If this field is null or has a group id of ANYGID (defined in **/usr/include/lpss.h**), then users from any group may use the queue. *Bandef* is a single letter (either 'q', 'b', 'Q', or 'B') indicating the following.

q -- normal queue, no banner
b -- normal queue, banner page before every job
Q -- default queue, no banner
B -- default queue, banner page before every job

A default queue may accessed from lpr without specifying queue on the command line. Only one default queue is permissable per group. *p1* and the following optional fields specify the name of a printer in the *printers*(5) file which is also assigned to this group. Several printers may be specified, however no printer may be specified on more than one line.

FILES

/usr/lpd/.qmap

NAME

sccsfile − format of SCCS file

DESCRIPTION

An SCCS 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 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 (optional) contain comments associated with the delta.

The @e line ends the delta table entry.

User names. The login names of users who may add deltas to the file, separated by newlines. The lines containing these login names are surrounded by the bracketing lines "@u" and "@U". An empty list of user names allows anyone to make a delta.

Flags. Keywords used internally (see *admin* (1S) 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
@f b
@f m    <module name>
@f f    <floor>
@f c    <ceiling>
@f d    <default-sid>
@f n
```

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.

Comments. Arbitrary text 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 don't 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

get(1S), delta(1S), admin(1S), prt(1S)
SCCS/PWB User's Manual by L. E. Bonanni and A. L. Glasser.

NAME

tp − magnetic tape format

DESCRIPTION

The command *tp*(1) dumps files to and extracts files from magtape. Block zero contains a copy of a stand-alone bootstrap program.

Blocks 1 through 62 contain a directory of the tape. There are 496 entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

```
struct  tpent   {
        char    pathnam[32];
        short   uid;
        char    uid;
        char    gid;
        char    spare;
        char    size0;
        short   size2;
        long    time;
        short   tapea;          /* tape address */
        short   unused[8];
        short   cksum;          /* check sum */
}
```

The path name entry is the path name of the file when put on the tape. If the pathname starts with a zero word, the entry is empty. It is at most 32 bytes long and ends in a null byte. Mode, uid, gid, size and time modified are the same as described under i-nodes (*fs*(5)). The tape address is the tape block number of the start of the contents of the file. Every file starts on a block boundary. The file occupies (size+511)/512 blocks of continuous tape. The checksum entry has a value such that the sum of the 32 words of the directory entry is zero.

Blocks 63 on are available for file storage.

A fake entry has a size of zero. See *tp*(1).

SEE ALSO

tp(1), fs(5)

NAME

utmp, wtmp − utmp and wtmp entry formats

DESCRIPTION

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

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

/*      <sys/types.h> must be included.                                 */

#define     UTMP_FILE    "/etc/utmp"
#define     WTMP_FILE    "/etc/wtmp"

struct utmp
 {
        char ut_user[8] ;               /* User login name */
        char ut_id[2] ;                 /* /etc/lines id(usually line #) */
        char ut_line[12] ;              /* device name (console, lnxx) */
        short ut_pid ;                  /* process id */
        struct exit_status
         {
          char e_termination ;          /* Process termination status */
          char e_exit ;                 /* Process exit status */
         }
        ut_exit ;                       /* The exit status of a process
                                         * marked as DEAD_PROCESS.
                                         */
        short ut_type ;                 /* type of entry */
        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     UTMAXTYPE   DEAD_PROCESS        /* Largest legal value of ut_type */

/*      Special strings or formats used in the "ut_line" field when     */
/*      accounting for something other than a process.                  */
/*      ** Note **  each message is such that is takes exactly 11       */
/*      spaces + a null, so that it fills the "ut_line" array.          */

#define     RUNLVL_MSG  "run_level_%c"
#define     BOOT_MSG    "system_boot"
#define     OTIME_MSG   "old_time  "
#define     NTIME_MSG   "new_time  "
```

FILES

/usr/include/utmp.h
/etc/utmp
/etc/wtmp

SEE ALSO

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