NAME

a.out – assembler and link editor output

SYNOPSIS

#include <a.out.h>

DESCRIPTION

A.out is the output file of the assembler *as*(1) and the link editor *ld*(1). Both programs make *a.out* executable if there were no errors and no unresolved external references. Layout information as given in the include file for the VAX-11 is:

```
/*
 * Format of an a.out header
 */

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 uninitialized data */
        unsigned a_syms;    /* size of symbol table */
        unsigned a_entry;   /* entry point */
        unsigned a_trsize;  /* size of text relocation */
        unsigned a_drsize;  /* size of data relocation */
};

#define A_MAGIC1        0407/* normal */
#define A_MAGIC2        0410/* read-only text */
#define A_MAGIC3        0411/* separated I&D */
#define A_MAGIC4        0405/* overlay */

struct  nlist { /* symbol table entry */
        char    n_name[8]; /* symbol name */
        char    n_type;    /* type flag */
        char    n_other;
        short   n_desc;
        unsigned n_value;  /* value */
};

        /* values for type flag */
#define N_UNDF 0        /* undefined */
#define N_ABS  02       /* absolute */
#define N_TEXT 04       /* text */
#define N_DATA 06       /* data */
#define N_BSS  08
#define N_TYPE 037
#define N_FN   037      /* file name symbol */

#define N_GSYM 0040     /* global sym: name,,type,0 */
#define N_FUN  0044     /* function: name,,linenumber,address */
#define N_STSYM 0046    /* static symbol: name,,type,address */
#define N_RSYM 0100     /* register sym: name,,register,offset */
#define N_SLINE 0104    /* src line: ,,linenumber,address */
#define N_SSYM 0140     /* structure elt: name,,type,struct_offset */
```

```
#define N_SO    0144    /* source file name: name,,,address */
#define N_LSYM 0200     /* local sym: name,,type,offset */
#define N_SOL   0204    /* #line source filename: name,,,address */
#define N_PSYM 0240     /* parameter: name,,type,offset */
#define N_LBRAC         0300/* left bracket: ,,nesting level,address */
#define N_RBRAC         0340/* right bracket: ,,nesting level,address */
#define N_LENG 0376     /* second stab entry with length information */

#define N_EXT   01      /* external bit, or'ed in */

#define FORMAT          "%08x"

#define STABTYPES       0340
```

The file has four sections: a header, the program text and data, relocation information, and a symbol table (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*(1).

In the header the sizes of each section are given in bytes. The size of the header is not included in any of the other sizes.

When an *a.out* file is loaded into core for execution, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at 0 in the core image; the header is not loaded. If the magic number in the header is 0407(8), 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 0410, the data segment begins at the first 0 mod 512 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.

The stack will occupy the highest possible locations in the core image: growing downwards from 80000000(16). The stack is automatically extended as required. The data segment is only extended as requested by *break*(2).

The start of the text segment in the file is 32(10); the start of the data segment is 32+a_text; the start of the text relocation is 32+a_text+a_data; the start of the data relocation is 32+a_text+a_data+a_trsize; the start of the symbol table is 32+a_text+a_data+a_trsize+a_drsize.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file.

If a 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 byte in the text or data which is not a portion of a reference to an undefined external symbol is exactly that value which will appear in memory when the file is executed. If a byte in the text or data involves a reference to an undefined external symbol, as indicated by the relocation information, then the value 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 to the bytes in the file.

If relocation information is present, it amounts to six bytes per relocatable dataum. There is no relocation information if a_trsize+a_drsize==0. The relocation information is structured as

```
struct relocation_info {
        long address;       /* relative to current segment */
        long symbolnum:24,
```

A.OUT (5)

```
                        /* if extern then symbol table */
                        /* ordinal (0, 1, 2, ...) else */
                        /* segment number (same as symbol types) */
        pcrel:1, /* if so, segment offset has already */
                        /* been subtracted */
        length:2,       /* 0=byte, 1=word, 2=long */
        extern:1,       /* does not include value */
                        /* of symbol referenced */
        offset:1;       /* already includes origin */
                        /* of this segment (?) */
};
```

SEE ALSO

as(1), ld(1), nm(1)

NAME

acct – execution accounting file

SYNOPSIS

#include <sys/acct.h>

DESCRIPTION

Acct(2) causes entries to be made into an accounting file for each process that terminates. The accounting file is a sequence of entries whose layout, as defined by the include file is:

```
typedef unsigned short comp_t;
        /* "floating pt": 3 bits base 8 exp, 13 bits fraction */

struct  acct
{
        char    ac_comm[10]; /* command name */
        comp_t  ac_utime;    /* user time */
        comp_t  ac_stime;    /* system time */
        comp_t  ac_etime;    /* elapsed time */
        time_t  ac_btime;    /* beginning time */
        short   ac_uid;      /* user ID */
        short   ac_gid;      /* group ID */
        short   ac_mem;      /* average memory usage */
        comp_t  ac_io;       /* number of disk IO blocks */
        dev_t   ac_tty;      /* control typewriter */
        char    ac_flag;     /* accounting flag */
};
        /* flag bits */
#define AFORK 01             /* has executed fork, but no exec */
#define ASU 02               /* used super-user privileges */
```

If the process does an *exec*(2), the first 10 characters of the filename appear in *ac_comm*. The accounting flag contains bits indicating whether *exec*(2) was ever accomplished, and whether the process ever had super-user privileges.

SEE ALSO

acct(2), sa(1)

AR (5)

NAME

ar – archive (library) file format

SYNOPSIS

#include <ar.h>

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 and header layout as described in the include file are:

```
#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;
};
```

The name is a null-terminated string; the date is in the form of *time*(2); the user ID and group ID are numbers; the mode is a bit pattern per *chmod*(2); the size is counted in bytes.

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), nm(1)

BUGS

Coding user and group IDs as characters is a botch.

NAME

core — format of memory image file

DESCRIPTION

UNIX writes out a memory 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 memory image is called 'core' and is written in the process's working directory (provided it can be; normal access controls apply).

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

SEE ALSO

adb(1), signal(2)

NAME

dir – format of directories

SYNOPSIS

#include <sys/types.h>
#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, *filsys*(5). The 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 and for the root directories of removable file systems. In the first case, there is no parent, and in the second, the system does not permit off-device references. Therefore in both cases '..' has the same meaning as '.'.

SEE ALSO

filsys(5)

NAME

dump, ddate − incremental dump format

SYNOPSIS

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

DESCRIPTION

Tapes used by *dump* and *restor*(1) contain:

a header record
two groups of bit map records
a group of records describing directories
a group of records describing files

The format of the header record and of the first record of each description as given in the include file <*dumprestor.h*> is:

```
#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          dinodec_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 records in a physical tape block. *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 map follows. This bit map has a one bit for each inode that was dumped.
TS_ADDR A subrecord of a file description. See *c_addr* below.
TS_END End of tape record.
TS_CLRI A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped.
MAGIC All header records have this number in *c_magic*.
CHECKSUM
Header records 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 number of this (512-byte) record.
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 record sum to *CHECKSUM*.
c_dinode This is a copy of the inode as it appears on the file system; see *filsys*(5).
c_count The count of characters in *c_addr*.
c_addr An array of characters describing the blocks of the dumped 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; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, *TS_ADDR* records will be scattered through the file, each one picking up where the last left off.

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

The structure *idates* describes an entry of the file */etc/ddate* where dump history is kept. The fields of the structure are:

id_name The dumped filesystem is '/dev/*id_nam*'.
id_incno The level number of the dump tape; see *dump*(1).
id_ddate The date of the incremental dump in system format see *types*(5).

FILES

/etc/ddate

SEE ALSO

dump(1), dumpdir(1), restor(1), filsys(5), types(5)

NAME

environ – user environment

SYNOPSIS

extern char **environ;

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, time, nice*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by ':'. *Login*(1) sets PATH=:/bin:/usr/bin.

HOME A user's login directory, set by *login*(1) from the password file *passwd*(5).

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as *nroff* or *plot*(1), which may exploit special terminal capabilities. See *term*(7) for a list of terminal types.

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), sh(1), term(7), login(1)

NAME

filsys, flblk, ino − format of file system volume

SYNOPSIS

```
#include <sys/types.h>
#include <sys/flbk.h>
#include <sys/filsys.h>
#include <sys/ino.h>
```

DESCRIPTION

Every file system storage volume (e.g. RF disk, RK disk, RP disk, DECtape reel) has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the *super block*. The layout of the super block as defined by the include file *<sys/filsys.h>* is:

```
        /* Structure of the super-block */
struct filsys {
        unsigned short s_isize;         /* first block not in 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 manipulation */
        char     s_ilock;               /* lock during i-list manipulation */
        char     s_fmod;                /* super block modified flag */
        char     s_ronly;               /* mounted read-only flag */
        time_t   s_time;                /* last super block update */
        /* remainder not maintained by this version of the system */
        daddr_t  s_tfree;               /* total free blocks*/
        ino_t    s_tinode;              /* total free inodes */
        short    s_m;                   /* interleave factor */
        short    s_n;                   /* " " */
        char     s_fname[6];            /* file system name */
        char     s_fpack[6];            /* file system pack name */
};
```

S_isize is the address of the first block after the i-list, which starts just after the super-block, in block 2. Thus the i-list is *s_isize*−2 blocks long. *S_fsize* is the address of the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block addresses; if an 'impossible' block address 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 NICFREE free block numbers. NICFREE is a configuration constant. *S_free[0]* is the block address of the head of a chain of blocks constituting the free list. The layout of each block of the free chain as defined in the include file *<sys/fblk.h>* is:

```
struct fblk
{
        int      df_nfree;
        daddr_t  df_free[NICFREE];
};
```

FILSYS (5)

The fields *df_nfree* and *df_free* in a free block are used exactly like *s_nfree* and *s_free* in the super block. To allocate a block: decrement *s_nfree,* and the new block number is *s_free[s_nfree].* If the new block address is 0, there are no blocks left, so give an error. If *s_nfree* became 0, read the new block into *s_nfree* and *s_free.* To free a block, check if *s_nfree* is NICFREE; 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 address and increment *s_nfree.*

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 inodes (up to NICINOD) into the *s_inode* array, then try again. To free an i-node, provided *s_ninode* is less than NICINODE, place its number into *s_inode[s_ninode]* and increment *s_ninode.* If *s_ninode* is already NICINODE, don't 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.

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 write-protection indicator; its disk value is also immaterial.

S_time is the last time the super-block of the file system was changed. During a reboot, *s_time* of the super-block for the root file system is used to set the system's idea of the time.

The fields *s_tfree, s_tinode, s_fname* and *s_fpack* are not currently maintained.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. I-nodes are 64 bytes long, so 8 of them fit into a block. 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. The format of an i-node as given in the include file <*sys/ino.h*> is:

```
        /* Inode structure as it appears on a disk block. */
struct dinode
{
        unsigned short di_mode;/* mode and type of file */
        short   di_nlink;       /* number of links to file */
        short   di_uid;         /* owner's user id */
        short   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 created */
};
#define INOPB 8                 /* 8 inodes per block */
/*
* the 40 address bytes:
*       39 used; 13 addresses
*       of 3 bytes each.
*/
```

Di_mode tells the kind of file; it is encoded identically to the *st_mode field of stat*(2). *Di_nlink* is the number of directory entries (links) that refer to this i-node. *Di_uid* and *di_gid* are the owner's user and group IDs. *Size* is the number of bytes in the file. *Di_atime* and *di_mtime* are the times of last access and modification of the file contents (read, write or create) (see *times*(2)); *Di_ctime* records the time of last modification to the inode or to the file, and is used to determine whether it should be dumped.

Special files are recognized by their modes and not by i-number. A block-type special file is one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files, the *di_addr* field is occupied by the device code (see *types*(5)). The device codes of block and character special files overlap.

Disk addresses of plain files and directories are kept in the array *di_addr* packed into 3 bytes each. The first 10 addresses specify device blocks directly. The last 3 addresses are singly, doubly, and triply indirect and point to blocks of 128 block pointers. Pointers in indirect blocks have the type *daddr_t* (see *types*(5)).

For block *b* in a file to exist, it is not necessary that all blocks less than *b* exist. A zero block number either in the address words of the i-node or in an indirect block indicates that the corresponding block has never been allocated. Such a missing block reads as if it contained all zero words.

SEE ALSO

icheck(1), dcheck(1), dir(5), mount(1), stat(2), types(5)

NAME

group – group file

DESCRIPTION

Group contains for each group the following information:

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

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

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

FILES

/etc/group

SEE ALSO

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

NAME

mpxio – multiplexed i/o

SYNOPSIS

#include <sys/mx.h>

#include <sgtty.h>

DESCRIPTION

Data transfers on mpx files (see *mpx*(2)) are multiplexed by imposing a record structure on the io 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 and M_BLK.

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_BLK – if non-blocking mode has been enabled on an mpx file descriptor *xd* by executing *ioctl(xd, MXNBLK, 0)*, write operations on the file are truncated in the kernel when internal queues become full. This is done on a per-channel basis: the parameter is a count of the number of characters not transferred to the

MPXIO (5)

channel on which M_BLK is received.

M_UBLK – is generated for a channel after M_BLK when the internal queues have drained below a threshold.

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 *<sgtty.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.

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 64 bytes long; the first 32 are the null-padded name of the place where the special file is mounted; the second 32 are the null-padded name of the special file. The special file has all its directories stripped away; that is, everything through the last '/' is thrown away.

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)

PASSWD (5)

NAME

passwd – password file

DESCRIPTION

Passwd contains for each user the following information:

name (login name, contains no upper case)
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 ID's to names.

FILES

/etc/passwd

SEE ALSO

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

NAME

plot – graphics interface

DESCRIPTION

Files of this format are produced by routines described in *plot*(3), and are interpreted for various devices by commands described in *plot*(1). 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*(3).

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*(1).

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 newline.

a arc: The first four bytes give the center, the next four give the starting point, and the last four give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise.

c circle: The first four bytes give the center of the circle, the next two the radius.

e erase: Start another frame of output.

f linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are 'dotted,' 'solid,' 'longdashed,' 'shortdashed,' and 'dotdashed.' Effective only in *plot 4014* and *plot ver*.

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*(1). 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 isn't square.

4014	space(0, 0, 3120, 3120);
ver	space(0, 0, 2048, 2048);
300, 300s	space(0, 0, 4096, 4096);
450	space(0, 0, 4096, 4096);

SEE ALSO

plot(1), plot(3), graph(1)

NAME

tp – DEC/mag tape formats

DESCRIPTION

The command *tp* dumps files to and extracts files from DECtape and magtape. The formats of these tapes are the same except that magtapes have larger directories.

Block zero contains a copy of a stand-alone bootstrap program. See *bproc*(8).

Blocks 1 through 24 for DECtape (1 through 62 for magtape) contain a directory of the tape. There are 192 (resp. 496) entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

```
struct {
        char path_name[32];
        unsigned short mode;
        char uid;
        char gid;
        char unused;
        char size[3];
        long time_modified;
        unsigned short tape_address;
        char unused1[16];
        unsigned short checksum;
};
```

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
(see file system
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 25 (resp. 63) on are available for file storage.

A fake entry
has a size of zero.

SEE ALSO

fs(5), tp(1)

TTYS (5)

NAME

ttys – terminal initialization data

DESCRIPTION

The *ttys* file is read by the *init* program and specifies which terminal special files are to have a process created for them which will allow people to log in. It contains one line per special file.

The first character of a line is either '0' or '1'; the former causes the line to be ignored, the latter causes it to be effective. The second character is used as an argument to *getty*(8), which performs such tasks as baud-rate recognition, reading the login name, and calling *login*. For normal lines, the character is '0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. (*Getty* will have to be fixed in such cases.) The remainder of the line is the terminal's entry in the device directory, /dev.

FILES

/etc/ttys

SEE ALSO

init(8), getty(8), login(1)

TYPES (5)

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 short      ino_t;
typedef long                time_t;
typedef int                 label_t[10];
typedef short               dev_t;
typedef long                off_t;
/* major part of a device */
#define major(x)           (int)(((unsigned)x>>8)&0377)

/* minor part of a device */
#define minor(x)           (int)(x&0377)

/* make a device number */
#define makedev(x,y)       (dev_t)(((x)<<8) | (y))
```

The form *daddr_t* is used for disk addresses except in an i-node on disk, see *filsys*(5). 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

filsys(5), time(2), lseek(2), adb(1)

NAME

utmp, wtmp – login records

SYNOPSIS

#include <utmp.h>

DESCRIPTION

The *utmp* file allows one to discover information about who is currently using UNIX. The file is a sequence of entries with the following structure declared in the include file:

```
struct utmp {
        char    ut_line[8];             /* tty name */
        char    ut_name[8];             /* user id */
        long    ut_time;                /* time on */
};
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of *time*(2).

The *wtmp* file records all logins and logouts. Its format is exactly like *utmp* except that a null user name indicates a logout on the associated terminal. Furthermore, the terminal name '~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with terminal names ' ' and '|' indicate the system-maintained time just before and just after a *date* command has changed the system's idea of the time.

Wtmp is maintained by *login*(1) and *init*(8). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by *ac*(1).

FILES

/etc/utmp
/usr/adm/wtmp

SEE ALSO

login(1), init(8), who(1), ac(1)

WTMP (5)

NAME

wtmp – user login history

DESCRIPTION

This file records all logins and logouts. Its format is exactly like *utmp*(5) except that a null user name indicates a logout on the associated typewriter. Furthermore, the typewriter name '~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with typewriter names ' ' and '}' indicate the system-maintained time just before and just after a *date* command has changed the system's idea of the time.

Wtmp is maintained by *login*(1) and *init*(8). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by *ac*(1).

FILES

/usr/adm/wtmp

SEE ALSO

utmp(5), login(1), init(8), ac(1), who(1)