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 header always contains 8 words:

1 A magic number (407, 410, or 411(8))
2 The size of the program text segment
3 The size of the initialized portion of the data segment
4 The size of the uninitialized (bss) portion of the data segment
5 The size of the symbol table
6 The entry location (always 0 at present)
7 Unused
8 A flag indicating relocation bits have been suppressed

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 (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 (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 11/45 handbook for restrictions which apply to this situation.

The stack will occupy the highest possible locations in the core image: from 177776(8) and growing downwards. The stack is automatically extended as required. The data segment is only extended as requested by the *break* 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. The next word is a flag indicating the type of symbol. The following values are possible:

00 undefined symbol
01 absolute symbol
02 text segment symbol
03 data segment symbol
37 file name symbol (produced by ld)
04 bss segment symbol
40 undefined external (.globl) symbol
41 absolute external symbol
42 text segment external symbol
43 data segment external symbol

44 bss segment external symbol

Values other than those given above may occur if the user has defined some of his own instructions.

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.

SEE ALSO

as(I), ld(I), strip(I), nm(I)

NAME

ar − archive (library) 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 177545(8) (it was chosen to be unlikely to occur anywhere else). The header of each file is 26 bytes long:

```
struct archive {
        char    a_name[14];     /* file name, null padded on right */
        long    a_date;  /* modification time of file */
        char    a_uid;   /* user ID of file owner */
        char    a_gid;   /* group ID of file owner */
        int     a_mode; /* file mode */
        long    a_size;  /* file 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.

FILES

/usr/include/archive.h

SEE ALSO

ar(I), ld(I)

NAME

ascii − map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, to be printed as needed. It contains:

```
|000 nul|001 soh|002 stx|003 etx|004 eot|005 enq|006 ack|007 bel|
|010 bs |011 ht |012 nl |013 vt |014 np |015 cr |016 so |017 si |
|020 dle|021 dc1|022 dc2|023 dc3|024 dc4|025 nak|026 syn|027 etb|
|030 can|031 em |032 sub|033 esc|034 fs |035 gs |036 rs |037 us |
|040 sp |041  ! |042  " |043  #|044  $|045  % |046  & |047  ´ |
|050  ( |051  ) |052  *|053  + |054  , |055  − |056  . |057  / |
|060  0|061  1|062  2|063  3|064  4|065  5|066  6|067  7|
|070  8|071  9|072  : |073  ; |074  < |075  = |076  > |077  ? |
|100  @ |101  A |102  B |103  C |104  D |105  E |106  F |107  G |
|110  H |111  I |112  J |113  K |114  L |115  M |116  N |117  O |
|120  P |121  Q |122  R |123  S |124  T |125  U |126  V |127  W |
|130  X |131  Y |132  Z |133  [ |134  \ |135  ] |136  ^ |137  _|
|140  ` |141  a |142  b|143  c |144  d|145  e |146  f |147  g|
|150  h|151  i |152  j |153  k|154  l |155  m |156  n|157  o|
|160  p|161  q|162  r |163  s |164  t |165  u|166  v|167  w |
|170  x|171  y|172  z |173  { |174  | |175  } |176  ~ |177 del|
```

FILES

found in /usr/pub

NAME

checklist − list of file systems processed by check

DESCRIPTION

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

SEE ALSO

check(VIII)

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(II)* 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* . Currently for PWB/UNIX systems it is 768 bytes. The remainder represents the actual contents of the user’s core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the *user* structure of the system. The important stuff not detailed therein is the locations of the registers. Here are their offsets. The parenthesized numbers for the floating registers are used if the floating-point hardware is in single precision mode, as indicated in the status register.

fpsr	0004
fr0	0006 (0006)
fr1	0036 (0022)
fr2	0046 (0026)
fr3	0056 (0032)
fr4	0016 (0012)
fr5	0026 (0016)

The following registers are located relative to end of the first section.

r0	−6
r1	−12
r2	−30
r3	−26
r4	−24
r5	−22
sp	−14
pc	−4
ps	−2

In general the debuggers *db(I)* and *cdb(I)* are sufficient to deal with core images.

SEE ALSO

adb(I), cdb(I), db(I), signal(II)

NAME

cpio − format of cpio archive

SYNOPSIS

```
struct {
        int     hmagic,
                hdev,
                hino,
                hmode,
                huid,
                hgid,
                hnlink,
                hmajmin;
        long    hmtime;
        int     hnamesize;
        long    hfilesize;
        char    hname[hnamesize rounded to word];
        char    data[hfilesize rounded to word];
} archive;
```

DESCRIPTION

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 *hmagic* contains the constant 070707. The items *hdev* through *hmtime* have meanings explained in *stat*(II). The length of the null-terminated pathname*hname* , including the null byte, is given by *hnamesize.*

The last record of the *archive* always contains the name ‘TRAILER!!!’. Special files, directories, and the trailer are recorded with hfilesize = 0.

SEE ALSO

cpio(I), stat(II)

BUGS

This format should be reconciled with *archive*(V).

NAME

directory − 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. Directory entries are 16 bytes long. The first word is the i-number of the file represented by the entry, if non-zero; if zero, the entry is empty.

Bytes 2-15 represent the (14-character) file name, null padded on the right. These bytes are not cleared for empty slots. The structure is:

```
struct dir {
        int     d_ino;    /* i-number */
        char    d_name[14];       /* file name */
};
```

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 "**.**".

FILES

/usr/include/dir.h

SEE ALSO

fs(V)

NAME

dump − incremental dump tape format

DESCRIPTION

The *dump* (VIII) and *restor* (VIII) commands are used to write and read incremental dump magnetic tapes.

The dump tape consists of blocks of 512-bytes each. The first block has the following structure.

```
struct {
        int     isize;
        int     fsize;
        int     date[2];
        int     ddate[2];
        int     tsize;
};
```

Isize and *fsize* are the corresponding values from the super block of the dumped file system (see *fs* (V)). *Date* is the date of the dump. *Ddate* is the incremental dump date. The incremental dump contains all files modified between *ddate* and *date.* *Tsize* is the number of blocks per reel. This block checksums to the octal value 031415.

Next there are enough whole tape blocks to contain one word per file of the dumped file system. This is *isize* divided by 16 rounded to the next higher integer. The first word corresponds to i-node 1, the second to i-node 2, and so forth. If a word is zero, then the corresponding file exists, but was not dumped. (Was not modified after *ddate*.) If the word is −1, the file does not exist. Other values for the word indicate that the file was dumped and the value is one more than the number of blocks it contains.

The rest of the tape contains for each dumped file a header block and the data blocks from the file. The header contains an exact copy of the i-node (see *fs* (V)) and also checksums to 031415. The next-to-last word of the block contains the tape block number, to aid in (unimplemented) recovery after tape errors. The number of data blocks per file is directly specified by the control word for the file and indirectly specified by the size in the i-node. If these numbers differ, the file was dumped with a ‘phase error’.

SEE ALSO

dump(VIII), restor(VIII), fs(V)

NAME

ebcdic − file format

DESCRIPTION

The ebcdic format is a convenient representation for files consisting of card images in an arbitrary code. Files created by the *send*(I) command, to be entered into rje *xmit** queues, are in this format. So are files of punch output from HASP.

An ebcdic file is a simple concatenation of card records. A card record consists of a single control byte followed by a variable number of data bytes. The control byte specifies the number (which must lie in the range 0-80) of data bytes that follow. The data bytes are 8-bit codes that constitute the card image. If there are fewer than 80 data bytes, it is understood that the remainder of the card image consists of trailing blanks.

SEE ALSO

send(I), hasp(VIII)

NAME

fs − format of file system volume

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 256 word (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.* Starting from its first word, the format of a super-block is

```
struct    {
          unsigned    int       isize;
          unsigned    int       fsize;
                      int       nfree;
          unsigned    int       free[100];
                      int       ninode;
          unsigned    int       inode[100];
                      char      flock;
                      char      ilock;
                      char      fmod;
                      char      ronly;
          long        int       time;
                      int       pad[40];
          unsigned    int       tfree;
          unsigned    int       tinode;
                      char      fname[6];
                      char      fpack[6];
};
```

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*fr ee* array contains, in *free[1], ... , free[nfree−1],* up to 99 numbers of free blocks. *F ree[0]* is the block number of the head of a chain of blocks constituting the free list. The first word in each free-chain block is the number (up to 100) of free-block numbers listed in the next 100 words of this chain member. The first of these 100 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*nfr ee* 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 100 words into the *free* array. To free a block, check if *nfree* is 100; 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, 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.

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 32 bytes long, so 16 of them fit into a block. Therefore, i-node *i* is located in block (*i* + 31) / 16, and begins 32•((*i* + 31) (mod 16)) bytes from its start. I-node 1 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 is as follows.

```
struct {
        int     flags;          /* +0: see below */
        char    nlinks;         /* +2: number of links to file */
        char    uid;            /* +3: user ID of owner */
        char    gid;            /* +4: group ID of owner */
        char    size0;          /* +5: high byte of 24-bit size */
        int     size1;          /* +6: low word of 24-bit size */
        int     addr[8];        /* +8: block numbers or device number */
        int     actime[2];      /* +24: time of last access */
        int     modtime[2];     /* +28: time of last modification */
};
```

The flags are as follows:

```
100000  i-node is allocated
060000  2-bit file type:
        000000  plain file
        040000  directory
        020000  character-type special file
        060000  block-type special file.
010000  large file
004000  set user-ID on execution
002000  set group-ID on execution
000400  read (owner)
000200  write (owner)
000100  execute (owner)
000070  read, write, execute (group)
000007  read, write, execute (others)
```

Special files are recognized by their flags and not by i-number. A block-type special file is basically 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 high byte of the first address word specifies the type of device; the low byte specifies one of several devices of that type. The device type numbers of block and character special files overlap.

The address words of ordinary files and directories contain the numbers of the blocks in the file (if it is small) or the numbers of indirect blocks (if the file is large). Byte number *n* of a file is accessed as follo ws. *N* is divided by 512 to find its logical block number (say *b*) in the file. If the file is small (flag 010000 is 0), then *b*

must be less than 8, and the physical block number is *addr[b]*.

If the file is large, *b* is divided by 256 to yield *i*. If *i* is less than 8, then *addr[i]* is the physical block number of the indirect block. The remainder from the division yields the word in the indirect block which contains the number of the block for the sought-for byte.

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.

FILES

/usr/include/filsys.h
/usr/include/stat.h

SEE ALSO

icheck(VIII), dcheck(VIII)

NAME

fspec − format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on UNIX with non-standard tabs, *i.e.,* tabs which are not set at the simple interval of eight columns. Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

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

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

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

Standard tabs are specified by 't−8' or, equivalently, 't1,9,17,25,etc'. The canned tabs which are recognized are defined by the *tabs*(I) command − a,a2,c,c2,c3,f,p,s,u.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

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

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

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

Default values, which are assumed for parameters not supplied, are '**t**−8' and '**m**0'. If the **s** parameter is not specified, no size checking is performed.

If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file.

The following is an example of a line containing a format specification:

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

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

Several *Programmer's Workbench* commands correctly interpret the format specification for a file. Among them is *gath* which may be used to convert files to a standard format acceptable to other UNIX commands.

SEE ALSO

ed(I), gath(I), reform(I), send(I), tabs(I)

NAME

greek − graphics for extended TELETYPE Model 37 type-box

SYNOPSIS

cat /usr/pub/greek

DESCRIPTION

Greek gives the mapping from ascii to the ‘‘shift out’’ graphics in effect between SO and SI on a TELETYPE® Model 37 with a 128-character type-box. It contains:

alpha	α	A	beta	β	B	gamma	γ	\
GAMMA	Γ	G	delta	δ	D	DELTA	Δ	W
epsilon	ε	S	zeta	ζ	Q	eta	η	N
THETA	Θ	T	theta	θ	O	lambda	λ	L
LAMBDA	Λ	E	mu	μ	M	nu	ν	@
xi	ξ	X	pi	π	J	PI	Π	P
rho	ρ	K	sigma	σ	Y	SIGMA	Σ	R
tau	τ	I	phi	ϕ	U	PHI	Φ	F
psi	ψ	V	PSI	Ψ	H	omega	ω	C
OMEGA	Ω	Z	nabla	∇	[	not	¬	_
partial	∂	]	integral	∫	^			

SEE ALSO

ascii(V)

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(I), login(I), crypt(III), passwd(I)

NAME

master − master device information table

DESCRIPTION

This file is used by the *config* (VIII) program to obtain device configuration information that enables it to generate the *low.s* and *conf.c* files.

The file consists of two parts, separated by a line with a dollar sign ($) in column 1. Part one contains device information, while part two contains names of devices that have aliases. Any line with an asterisk (*) in column 1 is treated as a comment.

Part one contains lines consisting of at least 10 fields and at most 13 fields, with the fields delimited by tabs and/or blanks, as follows:

Field 1: device name (8 characters maximum).

Field 2: interrupt vector size (decimal, in bytes).

Field 3: device mask − each ‘‘on’’ bit indicates that the handler exists, as follows:

000020	open handler
000010	close handler
000004	read handler
000002	write handler
000001	sgtty handler.

Field 4: device type indicator, as follows:

000020	immediate allocation
000010	block device
000004	character device
000002	floating vector
000001	fixed vector.

Field 5: handler prefix (4 characters maximum).

Field 6: device address size (decimal).

Field 7: major device number for block-type device.

Field 8: major device number for character-type device.

Field 9: maximum number of devices per controller (decimal).

Field 10: maximum bus request level (4 through 7).

Fields 11-13: optional configuration table structure declarations (8 characters maximum).

Part two contains lines with two fields each, as follows:

Field 1: alias of device (8 characters maximum).

Field 2: reference name of device (8 characters maximum, must have occurred in part one).

SEE ALSO

config(VIII)

NAME

mnttab − mounted file system table

DESCRIPTION

Mnttab resides in directory **/etc** and contains a table of devices mounted by the *mount*(VIII) command.

Each entry is 26 bytes in length; the first 10 bytes are the null-padded name of the place where the *special file* is mounted; the next 10 bytes represent the null-padded root name of the mounted special file; the remaining 6 bytes contain the mounted *special file's* read/write permissions and the date which it was mounted.

The maximum number of entries in *mnttab* is based on the system parameter, **NMOUNT,** located in **/sys/sys/cf/conf.c** which defines the number of allowable mounted special files.

SEE ALSO

mount(VIII), umount(VIII)

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
comment
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 comment field should identify the user, e.g., <dept #> name (account #). Each user is separated from the next by a newline. 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

login(I), crypt(III), passwd(I), group(V)

NAME

plot − graphics interface

DESCRIPTION

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

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. Not effective in *vt0*. See *plot*(I).

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. Effective only in *vt0*.

c circle: The first four bytes give the center of the circle, the next two the radius. Effective only in *vt0*.

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 *tek*.

d dot: Begin a horizontal dotted line at the point given by the next four bytes. The following two bytes are a signed x-increment, and the two after are a word count. The indicated number of words follow. A point is plotted for each 1-bit in the list, and skipped for each 0-bit. Each point is offset rightward by the x-increment. Effective only in *vt0*.

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*(I). 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.

tek	space(0, 3120, 0, 3120);
t300	space(0, 4096, 0, 4096);
t300s	space(0, 4096, 0, 4096);
t450	space(0, 4096, 0,4096)
vt0	space(0, 2048, 0, 2048);

SEE ALSO

plot(I), plot(III), graph(I)

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 ne wlines. 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. Each flag line takes the form:

@f <flag> <optional text>

There are, at present, only eight flags 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>

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.

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(I), delta(I), admin(I), prt(I)
SCCS/PWB User’s Manual by L. E. Bonanni and A. L. Glasser.

NAME

sha − Shell accounting file

DESCRIPTION

The file */etc/sha* is used by each Shell to record command execution data. This information is *not* used for charging, but is helpful for system tuning, command design, and monitoring of user activity. For each command executed, the Shell writes a 32-byte record of the following form:

```
struct {
        char    commandname[8];
        char    loginname[6];
        char    ttyletter;
        char    userid;
        long    date;
        long    realtime;
        long    cputime;
        long    systemtime;
} shrecord;
```

The *commandname* gives the last (or only) component of a pathname. When an asynchronously-executed command terminates, the Shell can obtain times, but not the actual command name. In this case, '**gok' is used. The name '()' indicates the completion of a parenthesized subshell.

The type (and therefore volume) of data recorded in */etc/sha* can be controlled by setting file permission bits appropriately. If it cannot be opened for writing, no data is recorded. Otherwise, the Shell tests the 3 bits of the group permission field to determine the kinds of recording to be done. If a Shell is reading from a TTY, it tests the high-order bit (04). If it is 0, the Shell records only external commands, i.e., those not built into the Shell. If the bit is 1, internal commands (such as *chdir*, =, etc.) are also recorded. A Shell that is not reading from a TTY uses the two low-order bits. If bit 02 is on, external commands are recorded. Setting bit 01 on adds internal commands. *Adm* should own */etc/sha*, and the group owner should be one not used elsewhere, such as 0. No data is ever recorded for the super-user. Sample file modes and their effects are:

606 Record external commands issued at TTY. This is the preferred mode.

666 Record everything but procedure-level internal commands, which can account for 30% of all command executions.

676 Record everything. This mode is probably of interest only to those who maintain the Shell. Be warned that this mode may cause */etc/sha* to grow by 1000 blocks per day in an active system.

SEE ALSO

sh(I), lastcom(VIII), sa(VIII)

NAME

tp − mag tape format

DESCRIPTION

The command *tp* dumps files to and extracts files from magtape.

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

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:

path name	32 bytes
mode	2 bytes
uid	1 byte
gid	1 byte
unused	1 byte
size	3 bytes
time modified	4 bytes
tape address	2 bytes
unused	16 bytes
check sum	2 bytes

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(V)). 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(I).*

SEE ALSO

fs(V), tapeboot(VIII), tp(I)

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 consists of lines of 3 characters each.

The first character is either ‘0’ or ‘1’; the former causes the line to be ignored, the latter causes it to be effective. The second character is the last character in the name of a terminal; e.g.*x* refers to the file ‘/dev/tty*x*’. The third character is used as an argument to the *getty* program, 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.)

FILES

/etc/ttys

SEE ALSO

init(VIII), getty(VIII), login(I)

NAME

utmp − user information

DESCRIPTION

This file allows one to discover information about who is currently using UNIX. The file is binary; each entry is 16(10) bytes long. The first eight bytes contain a user's login name or are null if the table slot is unused. The low order byte of the next word contains the last character of a terminal name. The next two words contain the user's login time. The last word is unused.

FILES

/etc/utmp

SEE ALSO

init(VIII) and login(I), which maintain the file; who(I), which interprets it.

NAME

wtmp − user login history

DESCRIPTION

This file records all logins and logouts. Its format is exactly like *utmp*(V) 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(I)* and *init(VIII)*. Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by*ac(VIII)*.

FILES

/usr/adm/wtmp

SEE ALSO

utmp(V), login(I), init(VIII), ac(VIII), who(I)