Chapter 4

File Formats

4.1 DESCRIPTION

This chapter outlines the formats of various files.

NAME

acct — per-process accounting file format

SYNOPSIS

```
#include <sys/acct.h>
```

DESCRIPTION

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

A typedef definition of the following type:

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

A structure *acct* with the following members:

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

A definition of the following symbolic names:

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

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

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

SEE ALSO

acct(2), exec(2), exit(2), fork(2).

APPLICATION USAGE

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

RELATIONSHIP TO SVID

The SVID, in Appendix K_EXT 3.0, "Header Files", defines only the typedef, the structure members and the list of symbols. All the other information derives from the equivalent UNIX System V Release 2.0 entry.

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 empty, 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

crypt(3C), passwd(4).

RELATIONSHIP TO SVID

This *group* file is not part of the SVID. It is identical to the UNIX System V release 2.0 definition, except that the SVID reads: "... field is null" in the third sentence of the second paragraph.

NAME

passwd — password file

DESCRIPTION

The file /etc/passwd contains, for each user, the following information:

1. name
2. encrypted password (may be empty)
3. numerical user ID
4. numerical group ID (may be empty)
5. reserved field
6. initial working directory
7. program to use as shell (may be empty)

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. Fields 2, 4 and 7 may be empty. However if they are not empty, they must be use for their stated purpose. Field 5 is a free field that is implementation specific. Fields beyond 7 are also free but may be standardized in the future. Each user's entry is separated from the next by a new-line.

This file resides in directory /etc. It has general read permission and can be used, for example, to map numerical user IDs to names.

Name is a character string that identifies a user. Its composition should follow the same rules as for file names.

If present, the *encrypted password* consists of 13 characters chosen from a 64-character alphabet (., /, 0-9, A-Z, a-z), except when the password is empty, in which case the encrypted password is also empty. Password aging is effected for a particular user if their encrypted password in the password file is followed by a comma and a non-null string of characters from the above alphabet.

By convention the last element in the pathname of the initial working directory is typically *name*.

FILES

/etc/passwd

SEE ALSO

crypt(3C).

RELATIONSHIP TO SVID

Identical to the SVID entry, in Appendix 2.9 "Passwd File Format", except that the SVID does not number the table, and uses the word "null" instead of "empty" when describing passwords in the DESCRIPTION.

NAME

utmp, wtmp — utmp and wtmp entry formats

SYNOPSIS

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

DESCRIPTION

These files, which hold user and accounting information, have the structure as defined by <**utmp.h**>. The header file declares the following symbolic names and structure members:

```
UTMP_FILE    /* pathname of utmp file */
WTMP_FILE    /* pathname of wtmp file */
ut_name
```

The structure *utmp* contains the following members:

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

The structure *exit_status* contains the following members:

```
short    e_termination;
short    e_exit;
```

Definitions for ut_type:

```
EMPTY
RUN_LVL
BOOT_TIME
OLD_TIME
NEW_TIME
INIT_PROCESS    /* Process spawned by "init" */
LOGIN_PROCESS   /* A "getty" process waiting for login */
USER_PROCESS    /* A user process */
DEAD_PROCESS
ACCOUNTING
UTMAXTYPE       /* Largest legal value of ut_type */
```

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

```
RUNLVL_MSG
BOOT_MSG
OTIME_MSG
NTIME_MSG
```

SEE ALSO

getut(3C).

APPLICATION USAGE

Chapter 1, Caveats, warns that the type of *ut_pid* may change from **short** as declared above. This will not cause problems to most applications provided that they do not use a **short** explicitly to hold the value of *ut_pid*; it is recommended that only **int** or longer variables are used for this purpose. No type-dependent problems will be caused if the field is accessed by name, as in the following example:

```
#include <stdio.h>
#include <sys/types.h>
#include <utmp.h>
main(){
        int pid;
        struct utmp *getutent(), *utp;

        while((utp = getutent()) != NULL){
                pid = utp—>ut_pid;
                printf("Value of pid = %d\n", pid);
                printf("Line %12s\n", utp—>ut_line);
        }
}
```

Problems will only occur if the field is accessed by reference (i.e. its address is taken).

SEE ALSO

getut(3C).

RELATIONSHIP TO SVID

These accounting information files are not part of the SVID. They are functionally equivalent to the UNIX System V Release 2.0 definition.

Chapter 5

Header Files

This chapter describes the contents of header files used for several system calls and library subroutines.

Header files contain the definition of symbolic constants, common structures, preprocessor macros and defined types, typedefs. The library routines in Chapters 2 and 3 specify the header files an application must include in order to use the routine. These files will be present on an applications development system; they do not have to be present on the target execution system.

This Chapter describes the following in each of the header files:

- The symbolic names, data types, data structures and macros defined in the header file that an application should use.
- The definition of the manifest constants.
- The system calls and library routines that may be used by an application and which reference the header file.

The header files <**termio.h**>, <**values.h**> and <**limits.h**> are not needed explicitly to support use of the routines defined in Chapters 2 and 3. They are included because they contain information that an application may need to use.

Relationships to the SVID are given as appropriate in the manual pages.

NAME

acct — per-process accounting records structures

SYNOPSIS

```
#include <sys/acct.h>
```

DESCRIPTION

This file is described in *acct*(4). Although also apropriate to mention it here, the reader is referred to *acct*(4); this ensures that there are not two pages containing the same information which the reader would have to check for consistency.

SEE ALSO

acct(4).

NAME

assert — verify program assertion

SYNOPSIS

```
#include <assert.h>
```

DESCRIPTION

This file contains the definitions used by *assert*(3X). A typical example of its contents is:

```
#ifdef NDEBUG
#define assert(EX)
#else
extern void _assert();
#define assert(EX) if (EX) ; else _assert("EX", __FILE__, __LINE__)
#endif
```

SEE ALSO

assert(3X)

RELATIONSHIP TO SVID

The SVID entry in Appendix BASE: 2.5, "Other Library Routines", refers users to *assert*(LIB) for the contents of this file.

NAME

ctype — character types

SYNOPSIS

#include <ctype.h>

DESCRIPTION

The following macros are defined in this file:

isalpha(c)	c is a letter
isupper(c)	c is an upper case letter
islower(c)	c is a lower case letter
isdigit(c)	c is a digit
isxdigit(c)	c is a hexadecimal digit [0-9], [A-F] or [a-f] .
isalnum(c)	c is an alphanumeric (letter or digit).
isspace(c)	c is a space, tab, carriage return, new-line, vertical tab or form-feed
ispunct(c)	c is a punctuation character (neither control nor alphanumeric).
isprint(c)	c is a printing character, code 040 (space) through 0176 (tilde).
isgraph(c)	c is a printing character, like *isprint* except false for space.
iscntrl(c)	c is a delete character (0177) or a ordinary control character (code less than 040)
isascii(c)	c is an ASCII character, code less than 0200
_toupper(c)	converts the lower case letter c to the corresponding upper case letter.
_tolower(c)	converts the upper case letter c to the corresponding lower case letter.
toascii(c)	converts c to an ASCII character by masking out high order bits.

APPLICATION USAGE

These all assume the ASCII character set is in use. *_toupper* and *_tolower* both give incorrect results if their arguments are not of the correct lower or upper case respectively.

SEE ALSO

conv(3C), ctype(3C)

RELATIONSHIP TO SVID

Identical to the SVID entry in Appendix BASE: 2.6, "Header Files".

NAME

PATH, HOME, TERM, TZ — user environment

DESCRIPTION

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

PATH — The sequence of directory prefixes that some applications apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (:). The logging-in or signing-on procedure will typically set *PATH=:/bin:/usr/bin*.

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

TERM — The kind of terminal for which output is to be prepared. This information is used by applications which want to exploit special capabilities of the terminal.

TZ — Time zone information. The format is xxx*n*zzz where *xxx* is standard local time zone abbreviation, *n* is the difference in hours from GMT, and *zzz* is the abbreviation for the daylight-saving local time zone, if any; for example, *MET-1EET*. See *ctime*(3C).

Further names may be placed in the environment by *exec*(2). It is unwise to conflict with certain variables that are frequently exported by widely used command interpreters: *MAIL*, *PS1*, *PS2*, *IFS*.

SEE ALSO

exec(2), ctime(3C).

FUTURE DIRECTIONS

The number in *TZ* will be defined as an optional minus sign followed by two hour digits and two minute digits, hhmm, in order to represent fractional time zones.

RELATIONSHIP TO SVID

The SVID gives the equivalent information in Appendix BASE: 2.8 Environmental Variables.

NAME

errno — system error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

The **<errno.h>** include file contains the following statements:

```
#include <sys/errno.h>
extern int errno;
```

The **<sys/errno.h>** system include file gives values for the following defined names.

N.B. The braces around the error names are a typographic convention, and do not form part of the name. The **#define** statements in the header file will look like this:

```
#define E2BIG  12345  /* perhaps.... */
```

and programs using these names will not include the braces.

[E2BIG]	Arg list too long
[EACCES]	Permission denied
[EAGAIN]	Resource unavailable, try again
[EBADF]	Bad file number
[EBUSY]	Device or resource busy
[ECHILD]	No child processes
[EDEADLK]	Deadlock avoided
[EDOM]	Math arg out of domain of func
[EEXIST]	File exists
[EFAULT]	Bad address†
[EFBIG]	File too large
[EINTR]	Interrupted system call
[EINVAL]	Invalid argument
[EIO]	I/O error
[EISDIR]	Is a directory
[EMFILE]	Too many open files
[EMLINK]	Too many links
[ENFILE]	File table overflow
[ENODEV]	No such device
[ENOENT]	No such file or directory
[ENOEXEC]	Exec format error
[ENOLCK]	No locks available
[ENOMEM]	Not enough space
[ENOSPC]	No space left on device
[ENOTBLK]	Block device required
[ENOTDIR]	Not a directory
[ENOTTY]	Not a character device

[ENXIO]	No such device or address
[EPERM]	Not owner
[EPIPE]	Broken pipe
[ERANGE]	Result too large
[EROFS]	Read only file system
[ESPIPE]	Illegal seek
[ESRCH]	No such process
[ETXTBSY]	Text file busy
[EXDEV]	Cross-device link

†The [EFAULT] error is caused by a program referencing data outside its legitimate address space. The reliable detection of this error cannot be guaranteed.

RELATIONSHIP TO SVID

Identical to the SVID entry in Appendix BASE 2.6 Header Files, in all the error definitions.

NAME

fcntl — file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

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

```
Flag values accessible to open(2) and fcntl(2)
(The first three can only be set by open)
O_RDONLY    /* Open for reading only */
O_WRONLY    /* Open for writing only */
O_RDWR      /* Open for reading and writing */
O_NDELAY    /* Non-blocking I/O */
O_APPEND    /* append (writes guaranteed at the end) */
O_SYNC      /* Synchronous write option */

Flag values accessible only to open(2)
O_CREAT     /* open with file create (uses third open arg)*/
O_TRUNC     /* open with truncation */
O_EXCL      /* exclusive open */

fcntl(2) requests
F_DUPFD     /* Duplicate fildes */
F_GETFD     /* Get fildes flags */
F_SETFD     /* Set fildes flags */
F_GETFL     /* Get file flags */
F_SETFL     /* Set file flags */
F_GETLK     /* Get blocking file locks */
F_SETLK     /* Set or clear file locks and fail on busy */
F_SETLKW    /* Set or clear file locks and wait on busy */
F_RDLCK     /* Read lock */
F_WRLCK     /* Write lock */
F_UNLCK     /* Remove lock(s) */
```

The structure *flock* contains the following members:

```
file segment locking control structure
short     l_type;   /* type of file */
short     l_whence; /* starting offset */
long      l_start;  /* relative offset */
long      l_len;    /* if 0 then until EOF */
int       l_pid;    /* returned with F_GETLK */
short     l_sysid;  /* returned with F_GETLK */
```

SEE ALSO

fcntl(2), open(2).

RELATIONSHIP TO SVID

Functionally equivalent to the SVID definition in Appendix BASE: 2.6 Header Files, with the addition of O_SYNC.

The SVID does not cater for synchronous writes.

The type of *l_pid* in struct *flock* has been corrected to **int**.

NAME

ftw — file tree walk

SYNOPSIS

```
#include <ftw.h>
```

DESCRIPTION

Codes for the third argument to the user-supplied function which is passed as the second argument to *ftw*:

```
FTW_F     /* file */
FTW_D     /* directory */
FTW_DNR   /* directory without read permission */
FTW_NS    /* unknown type, stat failed */
```

SEE ALSO

ftw(3C)

RELATIONSHIP TO SVID

Functionally equivalent to the SVID entry in Appendix BASE: 2.6 Header Files.

NAME

grp — group structure

SYNOPSIS

```
#include <grp.h>
```

DESCRIPTION

struct group contains the following members:

```
char    *gr_name;
char    *gr_passwd;
int     gr_gid;
char    **gr_mem;
```

SEE ALSO

getgrent(3).

RELATIONSHIP TO SVID

Not in the SVID.

NAME

limits — Implementation Specific Constants

SYNOPSIS

#include <limits.h>

DESCRIPTION

The following names are defined in <**limits.h**> and are used throughout the descriptive text of the X/OPEN System V specification. The values given in the column headed "Portability Value" are the values that applications should assume for portability across all X/OPEN systems. For example LOCK_MAX has a portability value of 32, which means that all X/OPEN systems will be capable of supporting up to 32 entries in their system lock tables. A particular system may be capable of supporting more than 32 entries, in which case its <**limits.h**> file will set LOCK_MAX to a higher value, but any application assuming this higher number is not guaranteed to be portable to all systems.

The items at the end of the list ending in "_MIN" give the most negative values that the mathematical types are guaranteed to be capable of representing. Numbers of a more negative value may be supported on some systems, as indicated by their <**limits.h**> file, but applications requiring such numbers are not guaranteed to be portable to all systems.

The symbol * in the Portability Value column indicates that there is no guaranteed value across all X/OPEN systems.

By inspecting the *limits.h* file on a specific system an applications writer can determine the actual limits in operation. Similarly, by *including* the file in the compilation an application can test the appropriate limits to determine whether it can operate on a particular system, or it may even alter its behaviour to match the system thus making itself portable across a varying range of limit settings/systems.

Name	Description	Portability Value
ARG_MAX	max length of argument to *exec(2)*	4096
CHAR_BIT	no. of bits in a **char**	8
CHAR_MAX	max integer value of a **char**	127
CHILD_MAX	max no. of processes per user id	4
CLK_TCK	no. of clock ticks per second	10
DBL_DIG	digits of precision of a **double**	*
FCHR_MAX	max size of a file in bytes	1,000,000
FLT_DIG	digits of precision of a **float**	*
FLT_MAX	max decimal value of a **float**	*
INT_MAX	max decimal value of an **int**	32767
LINK_MAX	max no. of links to a single file	8

LOCK_MAX	max no. of entries in system lock table	32
LONG_BIT	no. of bits in a long	32
LONG_MAX	max decimal value of a **long**	2,147,483,647
DBL_MAX	max decimal value of a **double**	*
MAX_CHAR	max size of character input buffer	*
NAME_MAX	max no. of characters in a file name	14
OPEN_MAX	max no. of files a process can have open	16
PASS_MAX	max no. of significant characters in a password	8
PATH_MAX	max no. of characters in a path name	255
PID_MAX	max value for a process ID	32,000
PIPE_BUF	max no. bytes atomic in write to a pipe	512
PIPE_MAX	max no. bytes written to a pipe in a write	4,096
PROC_MAX	max no. of simultaneous processes, system wide	8
SHRT_MAX	max decimal value of a **short**	32,767
SYSPID_MAX	max pid of system processes	1
STD_BLK	no. bytes in a physical IO block	256
SYS_NMLN	no. of chars in uname-returned strings	8
SYS_OPEN	max no. of files open on system	16
TMP_MAX	max no. of unique names generated by *tmpnam*(3S)	10000
UID_MAX	max value for a user or group ID	32,000
USI_MAX	max decimal value of an **unsigned**	65,535
WORD_BIT	no. of bits in a "word" or **int**	16
CHAR_MIN	min integer value of a **char**	0
DBL_MIN	min decimal value of a **double**	*
FLT_MIN	min decimal value of a **float**	*
INT_MIN	min decimal value of a **int**	-32,768
LONG_MIN	min decimal value of a **long**	-2,147,483,648
SHRT_MIN	min decimal value of a **short**	-32,768

SEE ALSO

Chapter 1 (Caveats: limits.h and values.h), values(5)

RELATIONSHIP TO SVID

The SVID does not define a <**limits.h**> include file and therefore does not allow for applications to determine the settings on a given system. It does use the listed names in place of absolute values in the descriptive text.

NAME

lock — process lock types

SYNOPSIS

#include <sys/lock.h>

DESCRIPTION

Values are given for the following flags for locking processes and texts:

PROLOCK	lock text and data segments into memory (process lock)
TXTLOCK	lock text segment into memory (text lock)
DATLOCK	lock data segment into memory (data lock)
UNLOCK	remove locks

SEE ALSO

plock(2).

RELATIONSHIP TO SVID

Functionally equivalent to the SVID. This include file is identified in Appendix K_EXT: 3.0 and its contents are given in *plock*(KEXT).

NAME

malloc — main memory allocator

SYNOPSIS

#include <malloc.h>

DESCRIPTION

This file provides definitions for constants defining *mallopt*, see *malloc*(3X) operations and declares the structure *mallinfo* to contain the members shown.

```
M_MXFAST    /* set size of blocks to be fast */
M_NLBLKS    /* set number of block in a holding block */
M_GRAIN     /* set number of sizes mapped to one, for
               small blocks */
M_KEEP      /* retain contents of block after a free until
               another allocation */
```

The structure *mallinfo* contains:

```
int arena;      /* total space in arena */
int ordblks;    /* number of ordinary blocks */
int smblks;     /* number of small blocks */
int hblks;      /* number of holding blocks */
int hblkhd;     /* space in holding block headers */
int usmblks;    /* space in small blocks in use */
int fsmblks;    /* space in free small blocks */
int uordblks;   /* space in ordinary blocks in use */
int fordblks;   /* space in free ordinary blocks */
int keepcost;   /* cost of enabling keep option */
```

SEE ALSO

malloc(3X).

RELATIONSHIP TO SVID

This is identical to the definition of *malloc.h* given in the SVID, Appendix BASE: 2.6 Header Files.

NAME

math — mathematical types

SYNOPSIS

#include <math.h>

DESCRIPTION

Values are given for the following useful constants:

M_E	value of *e*
M_LOG2E	value of base 2 log *e*
M_LOG10E	value of base 10 log *e*
M_LN2	value of base *e* log2
M_LN10	value of base *e* log10
M_PI	value of pi
M_PI_2	value of pi/2
M_PI_4	value of pi/4
M_1_PI	value of 1/pi
M_2_PI	value of 2/pi
M_2_SQRTPI	value of 2/sqrt(pi)
M_SQRT2	value of sqrt(2)
M_SQRT1_2	value of 1/sqrt(2)

The include file contains a define statement for the MAXFLOAT symbol which is machine dependent, and the value HUGE which is returned for error conditions found in the math library, see FUTURE DIRECTIONS, below.

MAXFLOAT	value of maximum floating point number
HUGE	error value returned by the math library

The structure *exception* is defined, containing the following members:

```
int type;       /* type of error that
                   ocurred */
char *name;     /* name of the function
                   that incurred the error */
double arg1;    /* argument 1 of the
                   invoked function */
double arg2;    /* argument 2 of the
                   invoked function */
double retval;  /* set of default values
                   returned by the function */
```

FUTURE DIRECTIONS

A macro HUGE_VAL will be defined to represent error values returned by the math functions. This macro will call a function which will either

return $+\infty$ on a system supporting IEEE P754 standard or +{MAXDOUBLE} on a system that does not support the IEEE P754 standard. The functions which currently return HUGE or $\pm$HUGE_VAL on overflow will return HUGE_VAL or $\pm$HUGE_VAL respectively.

SEE ALSO

erf(3M), exp(3M), floor(3M), gamma(3M), hypot(3M), sinh(3M), trig(3M).

RELATIONSHIP TO SVID

This is functionally equivalent to the SVID in Appendix BASE: 2.6, Header Files.

NAME

memory — memory operations

SYNOPSIS

#include <memory.h>

DESCRIPTION

This file declares the types of the functions performing memory operations.

SEE ALSO

memory(3C)

FUTURE DIRECTION

The declarations in **<memory.h>** will be moved to **<string.h>**.

RELATIONSHIP TO SVID

Functionally equivalent to the SVID in Appendix BASE: 2.6, Header Files.

NAME

mon — prepare execution profile

SYNOPSIS

```
#include <mon.h>
```

DESCRIPTION

The following structures are declared, and their members indicated:

```
struct hdr
char    *lpc;    /* low PC of range being profiled */
char    *hpc;    /* high PC of range being profiled */
int     nfns;    /* number of cnt structures */

struct cnt
char    *fnpc;   /* function PC */
long    mcnt;    /* call count */
```

A **typedef** is given for type *WORD*.

Definitions are given for the following names:

MON_OUT	filename for profile, e.g. "mon.out"
MPROGS0	
MSCALE0	
NULL	zero

SEE ALSO

monitor(3C).

RELATIONSHIP TO SVID

This include file is not defined in the SVID.

NAME

pwd — password file structure

SYNOPSIS

```
#include <pwd.h>
```

DESCRIPTION

Definitions are given for the following structures and their members.

struct passwd

```
char *pw_name;      /* name */
char *pw_passwd;    /* encrypted password (may be empty) */
int pw_uid;         /* numerical user ID */
int pw_gid;         /* numerical group ID (may be empty) */
char *pw_age;       /* password age */
char *pw_comment;
char *pw_gecos;     /* free field */
char *pw_dir;       /* initial working directory */
char *pw_shell;     /* program to use as shell (may be empty) */
```

struct comment

```
char *c_dept;
char *c_name;
char *c_acct;
char *c_bin;
```

SEE ALSO

getpwent(3C), putpwent(3C).

RELATIONSHIP TO SVID

This include file is not defined in the SVID.

NAME

search — hash search tables

SYNOPSIS

```
#include <search.h>
```

DESCRIPTION

Defines ENTRY as the structure *entry* through a typedef. *Entry* includes the following members:

```
char *key, *data;
```

Defines *ACTION* and *VISIT* as enumeration data types through typedefs as follows:

```
enum { FIND, ENTER } ACTION;

enum { preorder, postorder, endorder, leaf } VISIT;
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C), tsearch(3C)

RELATIONSHIP TO SVID

Identical to the SVID in Appendix BASE: 2.6 Header Files.

NAME

setjmp — stack environment type

SYNOPSIS

#include <setjmp.h>

DESCRIPTION

The include file contains a statement defining the symbolic constant *_JBLEN* which is machine dependent. A typedef is provided for type *jmp_buf*.

SEE ALSO

setjmp(3C)

RELATIONSHIP TO SVID

The SVID does not define this include file.

NAME

signal — signals

SYNOPSIS

#include <signal.h>

DESCRIPTION

The <signal.h> include file contains definitions of the following symbolic names:

SIGABRT	process abort signal
SIGALRM	alarm clock
SIGFPE	floating point exception
SIGHUP	hangup
SIGILL	illegal instruction (not reset when caught)*/
SIGINT	interrupt
SIGKILL	kill (cannot be caught or ignored)
SIGPIPE	write on a pipe with no one to read it
SIGQUIT	quit
SIGSEGV	segmentation violation
SIGSYS	bad argument to system call
SIGTERM	software termination signal from kill
SIGTRAP	trace trap (not reset when caught)
SIGUSR1	user defined signal 1
SIGUSR2	user defined signal 2

The include file contains a statement to define the symbolic constant NSIG which is machine dependent, and also the names SIG_DFL and SIG_IGN.

FUTURE DIRECTIONS

A macro SIG_ERR will be defined in <signal.h> to represent the return value (int(*())-1 by *signal*(2) in case of error.

APPLICATIONS USAGE

Refer back to Chapter 1 on portable signals.

RELATIONSHIP TO SVID

This is identical to the SVID definition in Appendix BASE: 2.6 Header Files, except that SIGSEGV has been addded from System V, Release 2.0 and SIGABRT is a future direction in the SVID.

NAME

stat — data returned by stat system call

SYNOPSIS

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

DESCRIPTION

The system calls *stat* and *fstat* return data whose structure is defined by this include file.

The structure *stat* contains the following members:

```
dev_t    st_dev;     /* device with directory entry for
                        file */
ino_t    st_ino;     /* inode number */
ushort   st_mode;    /* type of file (see below) */
short    st_nlink;   /* number of links */
ushort   st_uid;     /* user id of file owner */
ushort   st_gid;     /*group id of file owner */
dev_t    st_rdev;    /* device (if file is character or
                        block special) */
off_t    st_size;    /* file size in bytes */
time_t   st_atime;   /* time of last access */
time_t   st_mtime;   /* time of last data modification */
time_t   st_ctime;   /* time of last status change */
```

Symbolic names for the values of *st_mode* as well as macros to manipulate this field are also defined:

File type:

```
S_IFMT     /* type of file */
S_IFBLK    /* block special */
S_IFCHR    /* character special */
S_IFDIR    /* directory */
S_IFIFO    /* fifo special */
S_IFREG    /* regular */
```

File modes:

```
S_ISUID     /* set user id on execution */
S_ISGID     /* set group id on execution */
S_ISVTX‡    /* SEE APPLICATION USAGE BELOW */
S_IRWXU     /* read, write, execute/search by owner */
S_IRUSR     /* read permission, owner */
S_IWUSR     /* write permission, owner */
S_IXUSR     /* execute/search permission, owner */
S_IRWXG     /* read, write, execute/search by group */
S_IRGRP     /* read permission, group */
S_IWGRP     /* write permission, group */
```

```
S_IXGRP     /* execute/search permission, group */
S_IRWXO     /* read, write, execute/search by others */
S_IROTH     /* read permission, others */
S_IWOTH     /* write permission, others */
S_IXOTH     /* execute/search permission, others */

S_IREAD     /* read premission, owner */
S_IWRITE    /* write permission, owner */
S_IEXEC     /* execute/search premission, owner */
```

The following macros are defined:

```
S_ISBLK()   /* test for a block special file */
S_ISCHR()   /* test for a character special file */
S_ISDIR()   /* test for a directory */
S_ISFIFO()  /* test for a FIFO special file */
S_ISREG()   /* test for a regular file */
```

SEE ALSO

mknod(2), stat(2), types(5).

APPLICATIONS USAGE

Use of the macros is recommended for determining the type of a file.

It should be noted that S_IREAD, S_IWRITE and S_IEXEC are duplicated with the names S_IRUSR, S_IWUSR and S_IXUSR and the latter are preferred for portability as they are compatible with the /usr/group standard.

S_ENFMT: record locking enforcement is not currently part of the XVS but this name is reserved for future use.

‡S_ISVTX: Although this name is defined, the "save swapped text after use" functionality is becoming redundant and its use in not recommended.

RELATIONSHIP TO SVID

The *stat* structure is identical to that given in the SVID Appendix BASE 2.6: Header Files. The remainder is taken from the SVID FUTURE DIRECTION statement in Appendix BASE 1.6: Comparison to 1984 /usr/group Standard.

NAME

stdio — standard buffered input/output

SYNOPSIS

```
#include <stdio.h>
```

DESCRIPTION

Defines the following symbolic names:

```
BUFSIZ    /* size of stdio buffers */
NULL      /* NULL stdio pointer */
EOF       /* End of File return value */
stdin     /* File pointer to standard input */
stdout    /* File pointer to standard output */
stderr    /* File pointer to standard error output */
```

Defines the following data type through typedef:

FILE A structure containing information about a file.

APPLICATIONS USAGE

The following names may also be defined in this header file, and should only be used by applications developers in accordance with the definitions (where given) in other interface specifications.

L_ctermid	L_cuserid	L_tmpnam
P_tmpdir	_IOERR	_IOFBF
_IOLBF	_IOMYBUF	_IONBF
_IOREAD	_IORW	_IOWRT
_NFILE	_SBSIZE	_bufend
_bufsiz	clearerr	feof
ferror	fileno	getc
getchar	putc	putchar

SEE ALSO

bsearch(3C), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), hsearch(3C), lsearch(3C), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), stdio(3S), system(3S), tmpfile(3S), tmpnam(3S), tsearch(3C), ungetc(3S), vprintf(3S).

RELATIONSHIP TO SVID

Identical to the SVID entry in Appendix BASE 2.6: Header Files.

NAME

string — string operations

SYNOPSIS

#include <string.h>

DESCRIPTION

The types of the string functions are declared.

SEE ALSO

string(3C).

FUTURE DIRECTIONS

The SVID mentions that declarations in **<memory.h>** will be moved to **<string.h>**.

RELATIONSHIP TO SVID

Functionally identical to SVID in Appendix BASE 2.6: Header Files.

NAME

termio — define values for termio and ioctl

SYNOPSIS

#include <termio.h>

DESCRIPTION

This file contains the definitions used by *termio*(7) and *ioctl*(2). Refer to those descriptions for the structures and names defined.

SEE ALSO

ioctl(2), termio(7).

RELATIONSHIP TO SVID

The SVID entry in Appendix BASE: 2.6, Header Files, likewise refers users to *ioctl*(OS) and *termio*(DEV) for the contents of this file.

NAME

time — time types

SYNOPSIS

```
#include <time.h>
```

DESCRIPTION

The structure *tm* is declared, containing the following members:

```
int     tm_sec;
int     tm_min;
int     tm_hour;
int     tm_mday;
int     tm_mon;
int     tm_year;
int     tm_wday;
int     tm_yday;
int     tm_isdst;
```

SEE ALSO

ctime(3C).

RELATIONSHIP TO SVID

Identical to the SVID entry in Appendix BASE: 2.6 Header Files.

NAME

times — file access and modification times structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/times.h>
```

DESCRIPTION

The structure returned by *times*( ), *struct tms*, contains the following members:

```
time_t  tms_utime;      /* user time */
time_t  tms_stime;      /* system time */
time_t  tms_cutime;     /* user time, children */
time_t  tms_cstime;     /* system time, children */
```

SEE ALSO

times(2).

RELATIONSHIP TO SVID

The SVID defines <**sys/times.h**> only within the *times* (OS) description. It is not mentioned in Appendix BASE: 2.6 Header Files.

NAME

types — primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

Some data types used in system code are implementation dependent. These are defined in the **<types.h>** include file, which contains definitions for at least the following types:

daddr_t	Used for disk block addresses
ushort	Unsigned short
ino_t	Used for file serials numbers
time_t	Used for system time
dev_t	Used for device numbers
off_t	Used for file sizes

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.

SEE ALSO

ctime(3C), stat(2), times(2), utime(2).

RELATIONSHIP TO SVID

The type *daddr_t* is not defined in the SVID, the other types are defined in Appendix BASE: 2.6 Header Files. (The type *key_t* defined in Appendix K_EXT: 3.0 is used in shared memory, semaphore and message calls which are not included in the X/OPEN System V specification).

NAME

unistd — standard symbolic constants and structures

SYNOPSIS

```
#include <unistd.h>
```

DESCRIPTION

The file defines the symbolic constants and structures which are referenced elsewhere in the standard and which are not already defined or declared in some other "include" file. The contents of this file is shown below:

```
Symbolic constants for the "access" function:
R_OK        /* Test for "Read" Permission */
W_OK        /* Test for "Write" Permission */
X_OK        /* Test for "Execute"(Search) Permission */
F_OK        /* Test for existence of file */

Symbolic constants for the "lockf" function:
F_ULOCK     /* Unlock a previously locked region */
F_LOCK      /* Lock a region for exclusive use */
F_TLOCK     /* Test and lock a region for exclusive use */
F_TEST      /* Test a region for a previous lock */

Symbolic constants for the "lseek" function:
SEEK_SET    /* Set file pointer to "offset" */
SEEK_CUR    /* Set file pointer to current plus "offset" */
SEEK_END    /* Set file pointer to EOF plus "offset" */

Path names of the passwd and group files:
GF_PATH     /* Path name of the "group" file */
PF_PATH     /* Path name of the "passwd" file */
IF_PATH     /* Path name for <...> files */
```

SEE ALSO

fcntl(2), open(2).

RELATIONSHIP TO SVID

The SVID defines only F_ULOCK, F_LOCK, F_TLOCK and F_TEST. These are defined within the description of *lockf*(OS) and not in Appendix BASE: 2.6; the rest is described in Appendix BASE: 1.6, as a future direction.

NAME

ustat — file system statistics

SYNOPSIS

```
#include <ustat.h>
```

DESCRIPTION

struct ustat declares at least the following members:

```
daddr_t    f_tfree;      /* total free */
ino_t      f_tinode;     /* total inodes free */
char       f_fname[6];   /* filsys name */
char       f_fpack[6];   /* filsys pack name */
```

SEE ALSO

ustat(2).

RELATIONSHIP TO SVID

Identical to the SVID entry in Appendix BASE: 2.6 Header Files.

NAME

utmp — utmp file structure

SYNOPSIS

Refer to *utmp*(4).

SEE ALSO

utmp(4).

NAME

utsname — system name structure

SYNOPSIS

#include <sys/utsname.h>

DESCRIPTION

This file declares *struct utsname* which includes the following members:

```
char    sysname[{SYS_NMLN}];
char    nodename[{SYS_NMLN}];
char    release[{SYS_NMLN}];
char    version[{SYS_NMLN}];
char    machine[{SYS_NMLN}];
```

SEE ALSO

uname(2)

RELATIONSHIP TO SVID

Identical to the SVID in Appendix BASE: 2.6 Header Files.

NAME

values — machine-dependent values

SYNOPSIS

#include <values.h>

DESCRIPTION

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

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

BITSPERBYTE	The number of bits in a byte.
BITS(*type*)	The number of bits in a specified type (e.g., **int**).
HIBITS	The value of a short integer with only the high-order bit set (in most implementations, 0x8000).
HIBITL	The value of a long integer with only the high-order bit set (in most implementations, 0x80000000).
HIBITI	The value of a regular integer with only the high-order bit set (usually the same as HIBITS or HIBITL).
MAXSHORT	The maximum value of a signed short integer (in most implementations, 0x7FFF ≡ 32767).
MAXLONG	The maximum value of a signed long integer (in most implementations, 0x7FFFFFFF ≡ 2147483647).
MAXINT	The maximum value of a signed regular integer (usually the same as MAXSHORT or MAXLONG).
DMAXEXP	The maximum exponent of a double.
FMAXEXP	The maximum exponent of a float.
DMINEXP	The minimum exponent of a double.
FMINEXP	The minimum exponent of a float.
DMAXPOWTWO	The largest power of two exactly representable as a double.
FMAXPOWTWO	The largest power of two exactly representable as a float.

_FEXPLEN	The number of bits for the exponent of a float.
_EXPBASE	The exponent base.
_DEXPLEN	The number of bits for the exponent of a double.
_IEEE	1 if the IEEE standard representation is used.
_LENBASE	Number of bits in the exponent base.
HIDDENBIT	1 if high-significance bit in the mantissa is implicit. The largest power of two exactly representable as a double.
FSIGNIF	The number of significant bits in the mantissa of a single-precision floating-point number.
DSIGNIF	The number of significant bits in the mantissa of a double-precision floating-point number.
MAXFLOAT, LN_MAXFLOAT	The maximum value of a single-precision floating-point number, and its natural logarithm.
MAXDOUBLE, LN_MAXDOUBLE	The maximum value of a double-precision floating-point number, and its natural logarithm.
MINFLOAT, LN_MINFLOAT	The minimum positive value of a single-precision floating-point number, and its natural logarithm.
MINDOUBLE, LN_MINDOUBLE	The minimum positive value of a double-precision floating-point number, and its natural logarithm.

FILES

/usr/include/values.h

SEE ALSO

limits(5), math(5).

RELATIONSHIP TO SVID

Identical to the SVID in Appendix BASE: 2.6 Header Files, except that the names LN_MINFLOAT and LN_MAXFLOAT have been included from System V Release 2.0. It is not apparent why they are not part of the SVID.

NAME

varargs — handle variable argument list

SYNOPSIS

```
#include <varargs.h>

va_alist

va_dcl

void va_start(pvar)
va_list pvar;

type va_arg(pvar, type)
va_list pvar;

void va_end(pvar)
va_list pvar;
```

DESCRIPTION

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

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

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

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

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

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

va_end is used to clean up.

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

EXAMPLE

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

```
#include <varargs.h>
#define MAXARGS      100

/*      execl is called by
                execl(file, arg1, arg2, ..., (char *)0);
*/
execl(va_alist)
va_dcl
```

```
{
        va_list ap;
        char *file;
        char *args[MAXARGS];
        int argno = 0;

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

SEE ALSO

exec(2), printf(3S).

APPLICATION USAGE

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this from the stack frame. For example, *execl* is passed a zero pointer to signal the end of the list. *Printf* can tell how many arguments are there by the format.

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

RELATIONSHIP TO SVID

The SVID simply mentions the declaration, typedef or definition of the names *va_list, va_start, va_end, va_arg* and *va_decl* in Appendix BASE: 2.6 Header Files. The application writer is given no hint how to use them.

Chapter 6

This chapter is reserved for future use.

Chapter 7

Special Files

This chapter describes various special files which are present in all X/OPEN systems. Most systems will contain other entries for specific devices.

The entries shown here are mandatory on all systems, except for the source code transfer devices *sct*(7) which are only mandatory on systems with the relevant hardware.

Where there are corresponding entries in the SVID, they are to be found in Appendix 2.11, "Special Device Files".

NAME

console — System console interface

DESCRIPTION

/dev/console is a generic name given to the system console. It is usually linked to a particular machine dependent special file. It provides a basic I/O interface to the system console.

FILES

/dev/console.

SEE ALSO

termio(7).

RELATIONSHIP TO SVID

Identical to the SVID definition in Appendix 2.11, "Special Device Files", except that the SVID states that the system console works through the *termio* interface. This is not necessarily true of X/OPEN systems.

NAME

null — the null file

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

RELATIONSHIP TO SVID

Identical to the SVID definition in Appendix 2.11, "Special Device Files", except that the SVID also states that the "output of a command is written to the special file */dev/null* when the command is executed for its side effects and not for its output". This information is not relevant to applications development.

NAME

sctmt, sctfd — source code transfer devices (OPTIONAL)

DESCRIPTION

The files **/dev/sctmt{lmh}** and **/dev/sctfd{lm}** are the names of the special files used to transfer software between X/OPEN systems. Their descriptions follow.

½ Inch Magnetic Tape

The standard physical tape recording format is

- 9 track Phase Encoded (PE), 1600 bits per inch (bpi).

optional formats that may also be supported by particular systems in addition to this are

- 9 track Group Code Recording (GCR), 6250 bpi.
- 9 track Non Return to Zero Inverted (NRZI), 800 bpi.

Special File Names and Blocking

The device names associated with these formats are

name	format	blocksize(bytes)	remarks
/dev/sctmtl<number>	NRZI	512	optional
/dev/sctmtm<number>	PE	512	
/dev/sctmth<number>	GCR	512	optional
/dev/rsctmtl<number>	NRZI	see below	optional
/dev/rsctmtm<number>	PE	see below	
/dev/rsctmth<number>	GCR	see below	optional

Note that on the "raw" devices (rsct...), data is both read and written in blocks corresponding to the length requested in the read or write system call, see *read(2)* and *write(2)* in part 2 of the Guide.

The device names are usually links to system-specific device names.

Device Numbers

The part of the special file name described in the table above as <number> is constructed from the physical tape unit number with the addition of 0 or 128 (decimal) to indicate whether the tape is to be rewound on closure. Any tape that is opened for writing has a tape mark written on closure. Addition of 0 to the unit number causes the tape to be rewound to the beginning of tape mark (BOT); addition of 128 inhibits this.

Hosted implementations may need extra information to be specified in the device names, for example volume names.

5 ¼ inch Floppy Disk Exchange Physical Recording

The standard floppy disk recording formats are

Floppy Disc Recording Formats	
a)	80 tracks (96 tracks per inch) 2 tracks per cylinder 9 sectors per track 512 bytes per sector Modified Frequency Modulation (MFM) recording
b)	40 tracks (48 tracks per inch) 2 tracks per cylinder 8 sectors per track 512 bytes per sector Modified Frequency Modulation (MFM) recording read only

Note that 80 track is the preferred format; systems equipped only with 80 track drives will be able to read but not write 40 track disks.

Special File Names

The special file names associated with these formats are

Name	Number of Tracks
/dev/sctfdl<number>	40
/dev/sctfdm<number>	80

The device number is constructed from the physical drive number. To this is added 0 or 128 (decimal) to define whether cylinder 0 is accessible. 0 means that cylinder 0 is accessible, so the first sector accessed is sector 1 track 0 cylinder 0. 128 means that cylinder 0 is *not* accessible, so the first sector accessed is sector 1 track 0 cylinder 1.

FILES

/dev/sctmtm, /dev/sctfdm.

SEE ALSO

Part 6 of the Guide.

RELATIONSHIP TO SVID

This is not in the SVID. It is specific to X/OPEN systems.

NAME

termio — general terminal interface (OPTIONAL)

SYNOPSIS

#include <termio.h>

DESCRIPTION

This is the System V interface for asynchronous lines, irrespective of hardware involved. Refer to Chapter 1 for a discussion of the caveats with respect to this interface. The description below is relevant only to those lines configured to use this interface.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by the system and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *control terminal* for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork*(2). A process can break this association by changing its process group using *setpgrp*(2).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely full, or when an input line exceeds the maximum allowable number of input characters. Currently, this limit is {MAX_CHAR} characters. When the input limit is reached, all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, the character *'#'* erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the character *'@'* kills (deletes) the entire input line, and optionally outputs a new-line character. Both these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (*'\'*). In this case the escape character is not

read. The erase and kill characters may be changed.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (Rubout or ASCII DEL) generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location, see *signal*(2).

QUIT (Control-| or ASCII FS) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but the abnormal termination routines will be executed.

ERASE (#) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.

KILL (@) deletes the entire line, as delimited by a NL, EOF, or EOL character.

EOF (Control-d or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It can not be changed or escaped.

EOL (ASCII NUL) is an additional line delimiter, like NL. It is not normally used.

STOP (Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (Control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL may be redefined by the user. The ERASE, KILL, and EOF characters may be escaped by a preceding '\' character, in which case no special function is done.

When the carrier signal from the data-set drops, a *hang-up* signal is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hang-up signal is ignored, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

Several *ioctl*(2) system calls apply to terminal files. The primary calls use the *termio* structure, defined in <**termio.h**>:

A definition is given for:

```
NCC     /* size of the array
           * c_cc for special control characters */
```

The structure *termio* includes the following members:

```
unsigned        short   c_iflag;  /* input modes */
unsigned        short   c_oflag; /* output modes */
unsigned        short   c_cflag; /* control modes */
unsigned        short   c_lflag;  /* local modes */
char            c_line;           /* line discipline */
unsigned        char    c_cc[NCC];      /* control chars */
```

The special control characters are defined by the array *c_cc*. The relative positions and initial values for each function are as follows:

0	VINTR	DEL
1	VQUIT	FS
2	VERASE	'#'
3	VKILL	'@'
4	VEOF	EOT
5	VEOL	NUL
6	reserved	
7	SWTCH	

The *c_iflag* field describes the basic terminal input control:

IGNBRK	Ignore break condition.
BRKINT	Signal interrupt on break.
IGNPAR	Ignore characters with parity errors.
PARMRK	Mark parity errors.

INPCK	Enable input parity check.
ISTRIP	Strip character.
INLCR	Map NL to CR on input.
IGNCR	Ignore CR.
ICRNL	Map CR to NL on input.
IUCLC	Map upper-case to lower-case on input.
IXON	Enable start/stop output control.
IXANY	Enable any character to restart output.
IXOFF	Enable start/stop input control.

If IGNBRK is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if BRKINT is set, the break condition will generate an interrupt signal and flush both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

If PARMRK is set, a character with a framing or parity error which is not ignored is read as the three-character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error which is not ignored is read as the character NUL (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper-case alphabetic character is translated into the corresponding lower-case character.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read. If IXANY is set, any input character, will restart output which has been suspended.

If IXOFF is set, the system will transmit START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all-bits-clear.

The *c_oflag* field specifies the system treatment of output:

OPOST	Postprocess output.
OLCUC	Map lower case to upper on output.
ONLCR	Map NL to CR-NL on output.
OCRNL	Map CR to NL on output.
ONOCR	No CR output at column 0.
ONLRET	NL performs CR function.
OFILL	Use fill characters for delay.
OFDEL	Fill is DEL, else NUL.
NLDLY	Select new-line delays:
NL0	New-Line character type 0
NL1	New-Line character type 1
CRDLY	Select carriage-return delays:
CR0	Carriage-return delay type 0
CR1	Carriage-return delay type 1
CR2	Carriage-return delay type 2
CR3	Carriage-return delay type 3
TABDLY	Select horizontal-tab delays:
TAB0	Horizontal-tab delay type 0
TAB1	Horizontal-tab delay type 1
TAB2	Horizontal-tab delay type 2
TAB3	Expand tabs to spaces.
BSDLY	Select backspace delays:
BS0	Backspace-delay type 0
BS1	Backspace-delay type 1
VTDLY	Select vertical-tab delays:
VT0	Vertical-tab delay type 0
VT1	Vertical-tab delay type 1
FFDLY	Select form-feed delays:
FF0	Form-feed delay type 0
FF1	Form-feed delay type 1

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain

unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay. If OFILL is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL, otherwise NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

New-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the new-line delays. If OFILL is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If OFILL is set, delay type 1 transmits two fill characters, and type 2, four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters will be transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

The *c_cflag* field describes the hardware control of the terminal:

CBAUD	Baud rate:
B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud

B9600	9600 baud
B19200	19200 baud
B38400	38400 baud
CSIZE	Character size:
CS5	5 bits
CS6	6 bits
CS7	7 bits
CS8	8 bits
CSTOPB	Send two stop bits, else one.
CREAD	Enable receiver.
PARENB	Parity enable.
PARODD	Odd parity, else even.
HUPCL	Hang up on last close.
CLOCAL	Local line, else dial-up.
LOBLK	Block layer output.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal will not be asserted. Normally, this will disconnect the line. For any particular hardware, impossible speed changes are ignored.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stops bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used.

If CREAD is set, the receiver is enabled. Otherwise no characters will be received.

If HUPCL is set, the line will be disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. Otherwise modem control is assumed.

If LOBLK is set, the output of a job control layer will be blocked when it is not the current layer. Otherwise the output generated by that layer will be multiplexed onto the current layer.

The initial hardware control value after open is B300, CS8, CREAD, HUPCL.

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	Enable signals.
ICANON	Canonical input (erase and kill processing).
XCASE	Canonical upper/lower presentation.
ECHO	Enable echo.
ECHOE	Echo erase character as BS-SP-BS.
ECHOK	Echo NL after kill character.
ECHONL	Echo NL.
NOFLSH	Disable flush after interrupt or quit.

If ISIG is set, each input character is checked against the special control characters INTR, SWTCH, and QUIT. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g., 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read will not be satisfied until at least MIN characters have been received or the timeout value TIME has expired between characters. (See the "MIN/TIME Interaction Section" below). This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN and TIME values are stored in the position for the EOF and EOL characters, respectively. The time value represents tenths of seconds.

If XCASE is set, and if ICANON is set, an upper-case letter is accepted on input by preceding it with a '\' character, and is output preceded by a '\' character. In this mode, the following escape sequences are generated on output and accepted on input:

for:	*use*:
`	\'
|	\!
~	\^
{	\(
}	\)
\	\\

For example, *'A'* is input as \a, *'\n'* as \\n, and *'\N'* as \\\n.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which will clear the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character will be echoed after the kill character to emphasize that the line will be deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character will be echoed even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit, switch, and interrupt characters will not be done.

The initial line-discipline control value is all bits clear.

The primary *ioctl*(2) system calls have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETS — Get the parameters associated with the terminal and store in the *termio* structure referenced by **arg** (see the APPLICATION USAGE section below).

TCSETS — Set the parameters associated with the terminal from the structure referenced by **arg**. The change is immediate. (see the APPLICATION USAGE section below).

TCSETAW — Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.

TCSETAF — Wait for the output to drain, then flush the input queue and set the new parameters.

Additional *ioctl*(2) calls have the form:

```
ioctl (fildes, command, arg)
int arg;
```

The commands using this form are:

TCSBRK — Wait for the output to drain. If *arg* is 0, then send a break (zero bits for 0.25 seconds).

TCXONC — Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.

TCFLSH — If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

MIN/TIME Interaction

MIN represents the minimum number of characters that should be received when the read is satisfied (i.e., that is the characters are returned to the user). TIME is a timer of 0.1 second granularity that is used to timeout bursty and short term data transmissions. The four possible combinations of MIN and TIME and their interactions are described below.

A. MIN >0, TIME >0

In this case TIME serves as an intercharacter timer and is activated after the first character is received. It is reset upon receipt of each character. As soon as one character is received the intercharacter timer is started. If MIN characters are received befor the timer expires the read is satisfied. If the timer expires before MIN characters are received the characters received to that point are returned to the user.

B. MIN > 0, TIME = 0

Since the value of TIME is zero, the timer plays no role and only MIN is significant. In this case, the read is not satisfied until MIN characters are received.

C. MIN = 0, TIME > 0

Since MIN = 0, TIME no longer represents an intercharacter timer. It now serves as a read timer that is activated as soon as the *read*(2) call is processed (in canon). A read is satisfied as soon as a single character is received or the read timer expires, in which case the read will return with zero characters.

D. MIN = 0, TIME = 0

In this case the return is immediate. If characters are present they will be returned to the user.

FILES

/dev/tty*, termio.h

SEE ALSO

fork(2), ioctl(2), setpgrp(2), signal(2).

APPLICATION USAGE

TCGETA and are *ioctl*(2) commands that are reserved to maintain source code compatibility. Their use is even more system dependent than the *termio* interface and source code that uses these commands may not work correctly on all systems.

RELATIONSHIP TO SVID

Identical to the SVID entry in Appendix BASE 2.11, "Special Device Files" except for minor changes to the first paragraph.

NAME

tty — controlling terminal interface

DESCRIPTION

The file **/dev/tty** is, in each process, a synonym for the control terminal associated with the process group of that process, if any. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

FILES

/dev/tty.

RELATIONSHIP TO SVID

Identical to the SVID definition in Appendix 2.11, "Special Device Files".