NAME

ioctl − control device

SYNOPSIS

#include <sys/ioctl.h>

int ioctl (fildes, request, argp)
int fildes, request;
struct *argp;

DESCRIPTION

Ioctl manipulates the file or device indicated by *fildes* as specified by *request*. The requests and the kinds of things they can access are:

TIOCGETD, TIOCSETD
Get/set line discipline. *Argp* points to a structure containing an integer with a valid line discipline indicator integer.

TIOCHPCL
Hang up on last close. *Argp* indicates whether this feature should be turned on or off.

TIOCSETO, TIOCGETO
Get/set "other" bits. *Argp* contains a word with bits indicating which "other" bits are to be set/reset or interrogated. This request is essentially an extension of the old *stty/gtty* system call that allows transmission/response to xon/xoff, half duplex line, no-hangup, excluding future device opens, no sleeping if not ready, and non-standard tty escapes and kills.

TIOCGETP, TIOCSETP
These are equivalent to **gtty(fildes, argp)** and **stty(fildes, argp)**. They allow terminal (tty) characteristics to be set and examined. These include terminal input and output speed, the erase character and kill character, and mode flags. The allowed mode flags include hangup on last close, map tabs to spaces, upper case only, character echo, cr/lf mode, raw character input, parity, and delay on tabs, new lines, backspace, carriage return, and vt delay. Note that setting input speed to zero on a dh or dz line will disable the line by dropping the Data Terminal Ready(DTR) bit for the line.

TIOCSETN
Equivalent to old *stty* with *noflush.*

TIOCEXCL, TIOCNXCL
Get/clear the exclude bit, which disallows future opens on the device.

TIOCTSTP
Stop toggle transmit.

DIOCGETT, DIOCSETT
Get/set terminal parameters. These include terminal type, current cursor row and column (get only), variable row, last row, and terminal flags. The flags include special newline, auto newline on column 80, last column of last row special, echo of terminal cursor control, and not sending escape sequences to the user. It is used primarily for CRT terminals.

DIOCSETS
Set spy mode. All output directed to the terminal specified by *fildes* will be copied to the terminal of the process performing the *ioctl.* Only one spy operation may be active in the entire system at any time. The spy continues until explicitly turned off. Currently, spy is only effective on lines using the **STD_LTYPE** line discipline and is restricted to the super-user.

FIOCLEX, FIONCLEX
Set/clear auto close for a file. If auto close is set, then the file will not be passed to children across an *exec.*

FIOSPIPE, FIOGPIPE
Get/set pipe sleep flags. This enables/disables sleeping on reads/writes to a pipe, to avoid roadblocking. Normally, reads are blocking and writes are not.

VIOCGETD, VIOCSETD
Get/set versatec parameters.

There are also requests for the multiplexor (see *mpx*(2), *mpxio*(5) and <**sys/mx.h**>). In general, each line discipline has a unique header file which defines the line discipline number and format of the structure to be used with **DIOCGETP** and **DIOCSETP** requests.

The proper names for all these flags and other requests not currently used are contained in <**sys/ioctl.h**>, which is included here:

```
/*              @(#)ioctl.h     3.5             */
/*
 * structure of arg for ioctl TIOCSETP and TIOCGETP
 */
struct          ttiocb {
                char            ioc_ispeed;
                char            ioc_ospeed;
                char            ioc_erase;
                char            ioc_kill;
                short           ioc_flags;
};

/*
 * structure for old stty and gtty system calls.
 */
struct          sgttyb          {
                char            sg_ispeed;      /* input speed */
                char            sg_ospeed;      /* output speed */
                char            sg_erase;       /* erase character */
                char            sg_kill;        /* kill character */
                short           sg_flags;       /* mode flags */
};

/*
 * tty ioctl commands
 */
#define         TIOCGETD        (('t'<<8)|0)    /* get line discipline */
#define         TIOCSETD        (('t'<<8)|1)    /* set line discipline */
#define         TIOCHPCL        (('t'<<8)|2)    /* hangup on last close */
#define         TIOCMODG        (('t'<<8)|3)
#define         TIOCMODS        (('t'<<8)|4)
#define         TIOCSETO        (('t'<<8)|6)    /* set other bits */
#define         TIOCGETO        (('t'<<8)|7)    /* get other bits */
#define         TIOCGETP        (('t'<<8)|8)    /* gtty */
#define         TIOCSETP        (('t'<<8)|9)    /* stty */
#define         TIOCSETN        (('t'<<8)|10)   /* stty - no flush */
#define         TIOCEXCL        (('t'<<8)|13)   /* set exclude */
#define         TIOCNXCL        (('t'<<8)|14)   /* clr exclude */
#define         TIOCHMOD        (('t'<<8)|15)
#define         TIOCTSTP        (('t'<<8)|16)   /* toggle transmit stop */
#define         DIOCGETP        (('d'<<8)|8)    /* get discipline parameters */
#define         DIOCSETP        (('d'<<8)|9)    /* set discipline parameters */
#define         DIOCSETT        (('d'<<8)|10)   /* set terminal info */
```

```
#define         DIOCGETT    (('d'<<8)|11)   /* get terminal info */
#define         DIOCSETS    (('d'<<8)|12)   /* set spy mode */
#define         FIOCLEX                     (('f'<<8)|1)    /* set auto close */
#define         FIONCLEX    (('f'<<8)|2)    /* clr autoclose */
#define         FIOSPIPE    (('p'<<8)|1)    /* set pipe sleep flags */
#define         FIOGPIPE    (('p'<<8)|2)    /* get pipe sleep flags */
#define         VIOCGETD    (('v'<<8)|0)    /* Versatec */
#define         VIOCSETD    (('v'<<8)|1)    /* Versatec */

/*
 * Define standard line discipline for TIOCSETD and TIOCGETD
 */
#define         STD_LTYPE   (short)0
/*
 * Define half duplex line discipline for TIOCSETD and TIOCGETD
 */
#define HF_LTYPE            (short)4
/*
 * Format of third argument for TIOCSETD and TIOCGETD
 */
struct sgldisc {
                short       sgl_type;
};


/*
 * Following ioctl.h commands are used within the system only.
 */
#ifdef KERNEL
#define         OLDSGTTY    (('i'<<8)|1)
#define         GETRFP                      (('i'<<8)|2)
#define         GETWFP                      (('i'<<8)|3)
#endif

/*
 * Modes
 */
#define         HUPCL       01              /* hangup on last close */
#define         XTABS       02              /* map tabs to spaces on output */
#define         LCASE       04              /* upper case only terminal */
#define         ECHO        010             /* echo all received characters */
#define         CRMOD       020             /* map CR->LF;echo CR or LF as CR-LF */
#define         RAW         040             /* raw character input */
#define         ODDP        0100            /* odd parity rcvd/xmtd */
#define         EVENP       0200            /* even parity rcvd/xmtd */
#define         ANYP        0300            /* any parity mask */
#define         NLDELAY     001400
#define         TBDELAY     002000
#define         CRDELAY     030000
#define         VTDELAY     040000
#define         BSDELAY     0100000
#define         ALLDELAY    0173400

/*
 * Delay algorithms
 */
#define         CR0         0
#define         CR1         010000
#define         CR2         020000
#define         CR3         030000
#define         NL0         0
#define         NL1         000400
#define         NL2         001000
```

```
#define         NL3            001400
#define         TAB0           0
#define         TAB1           002000
#define         NOAL           004000
#define         FF0            0
#define         FF1            040000
#define         BS0            0
#define         BS1            0100000

/*
 * Speeds
 */
#define B0      0
#define B50     1
#define B75     2
#define B110    3
#define B134    4
#define B150    5
#define B200    6
#define B300    7
#define B600    8
#define B1200   9
#define B1800   10
#define B2400   11
#define B4800   12
#define B9600   13
#define EXTA    14
#define EXTB    15

/*
 * Character length and stop bits.
 * Character length does not include parity or stop bits.
 * Ored with ioc_ospeeed.
 */
#define         SETSTOP        0200            /* set to change stop or length bits */
#define         ONESTOP        0000
#define         TWOSTOP        0100            /* 1.5 stop bits at 75 baud */
#define         BITS5          0000
#define         BITS6          0020
#define         BITS7          0040
#define         BITS8          0060
#define         SLBITS         0160            /* Mask of stop and length bits */


/*
 * structure of arg for ioctl TIOCSETO and TIOCGETO
 */
struct ttiothcb {
                short          ioth_flags;
};

/*
 * Definition of "other" bits
 */
#define         TANDEMO        01              /* enable transmission of xon/xoff */
#define         HDPLX          0400            /* Half duplex line */
#define         NOHUP          01000           /* not dial device flag */
#define         XCLUDE         02000           /* disallow future opens */
#define         NOSLEEP        04000           /* dont sleep if nothing is ready */
#define         TANDEMI        040000          /* enable response to xon/xoff */
#define         STDTTY         0100000         /* non-standard tty escapes and kills */
```

```
/*
 * struct of arg for ioctl FIOSPIPE and FIOGPIPE
 */
struct          pipcb           {
                char            pip_rflg;       /* read flag; 0=>nosleep */
                char            pip_wflg;       /* write flag; 0=>nosleep */
};


/*
 * structure of ioctl arg for DIOCGETT and DIOCSETT
 */
struct          termcb          {
                char            st_flgs;        /* term flags */
                char            st_termt;       /* term type */
                char            st_crow;        /* gtty only - current row */
                char            st_ccol;        /* gtty only - current col */
                char            st_vrow;        /* variable row */
                char            st_lrow;        /* last row */
};

/*
 * Terminal types
 */
#define         TERM_NONE       0               /* tty */
#define         TERM_TEC        1               /* TEC Scope */
#define         TERM_V61        2               /* DEC VT61 */
#define         TERM_V10        3               /* DEC VT100 */
#define         TERM_TEX        4               /* Tektronix 4023 */
#define         TERM_D40        5               /* TTY Mod 40/1 */
#define         TERM_H45        6               /* Hewlitt-Packard 45 */
#define         TERM_D42        7               /* TTY Mod 40/2B */
#define TERM_C100               8               /* Concept 100*/

/*
 * Terminal flags
 */
#define TM_NONE                                 0000            /* use default flags */
#define TM_SNL                                  0001            /* special newline flag */
#define TM_ANL                                  0002            /* auto newline on column 80 */
#define TM_LCF                                  0004            /* last col of last row special */
#define TM_CECHO                0010            /* echo terminal cursor control */
#define TM_CINVIS               0020            /* do not send esc seq to user */
#define TM_SET                  0200            /* must be on to set/res flags */
```

Several of the modes and flags require further explanation:

LCASE Map upper case to lower case on input; map lower case to upper case on output. Map | to !; ‘ to ’; { to (; } to); ˜ to ˆ; \<C> to upper case input, where <C> is any upper case character.

RAW In raw mode, every character is immediately passed to the program without waiting for a full line to be typed. No input characters have a special meaning (e.g., the interrupt character DEL will not cause the program to be interrupted, but will be passed to the program as a character.). **LCASE** and **CRMOD** will still cause input mapping; output character processing is unaffected. If the transmitter has been stopped by the ESC key, setting **RAW** will release it. Note, however, that this can only be effective if the **TIOCSETP** command is utilized. Otherwise, the program will wait for the ESC key to be depressed again. Input and output data width is eight bits, but the eigth bit may be a parity bit depending upon the setting of **ODDP** and **EVENP**.

ODDP, EVENP
For the standard line discipline, a character will be rejected unless its parity matches that expected. If both bits are set, either parity is accepted and even parity is transmitted. If both bits are set and **RAW** is set, the parity is visible to and supplied by the user on input and output. If neither bit is set, no parity is expected and even parity is transmitted.

HDPLX For those communications controllers with the capability, disable reception during transmission.

XCLUDE When set, no one may open the line. Cleared upon the last close.

NOSLEEP
Return a zero if a read is performed and no characters are present. Don't wait to flush output on *close* or *ioctl.* Don't wait for carrier on the first *read* or *write* after an *open*, if carrier is not up. Normally, a process will block when waiting for carrier to come up after an *open.* This roadblock will take place in the first *read* or *write*, not the *open.*

STDTTY Change the erase character from # to _ and the delete line character from @ to $. In addition to CR and LF, wake up on / and !, and generate an interrupt upon reception of & or DEL.

TANDEMO
When set, transmission of xon/xoff is enabled. This turns off the keyboard when there are too many characters in the terminal hardware queue.

TANDEMI
When set, response to xon/xoff is enabled.

NOHUP Indicates that the line is not a dial-up line, and, therefore, will not hang up when the terminal session is completed.

DELAY For certain line speeds, a delay is desired for certain functions. Delay can be specified for CR, LF, tabs, backspaces, and formfeeds.

It is also possible for the user to set the number of data and stop bits, if the defaults are not satisfactory. The default is **TWOSTOP** at speeds B75 and B110, **ONESTOP** otherwise; **BITS5** for B75, with **BITS7** plus one bit even parity otherwise. These bits are or'd in with the *ioc_ospeed* flag. The **SETSTOP** bit must be set to change stop or length bits.

Normally, an **TIOCSETP** request will wait for output to be flushed before doing anything. This can be circumvented by using the **TIOCSETN** request.

The normal CB-UNIX line discipline is **STD_LTYPE**. Request **TIOCSETD** can be used to set the discipline to the commonly-supported half-duplex line discipline **HF_LTYPE**, and the transparent line discipline **TRANS_LTYPE**, described in <**sys/trans.h**>. Different line disciplines expect different values for certain modes. However, **STD_LTYPE** and **HF_LTYPE** require no additional information.

TRANS_LTYPE is a line discipline that allows the user full eight bit transparency on input and output with or without parity. For this line discipline, a *write* will perform no mapping. A *read* will return upon the occurrence of the first of the three conditions as specified by the user:

1) The requested number of characters have arrived.

2) The number of seconds, *ts_quanta*, has elapsed.

3) A break character has arrived.

If *ts_quanta* is zero, timing is disabled; otherwise, *ts_quanta* is the maximum wait time in seconds. If *ts_brk0* and *ts_brk1* are both zero, no break characters will awaken the process. If *ts_brk1* is 0377 then *ts_bbrk0* is taken as a single break character. Otherwise, both break characters are assumed valid. **NCDELAY, XTABS, LCASE, ECHO, CRMOD, RAW,** and **STDTTY** have no meaning for this line discipline.

The **DIOCSETT** request is used to specify the type of CRT connected to a line. **TERM_NONE** is the standard, non-CRT type. If a type other than **TERM_NONE** is specified, input and output mapping will occur for the CRT language defined in the header file **<sys/crtctl.h>**. In this case, the ESC character takes on special meaning, escaping the subsequent characters on input and output. The terminal flags *st_flgs* and modes are given a default set of values when the terminal type is set. The modes may be subsequently changed with a **DIOCSETT** request. The flags may be changed by setting the **TM_SET** bit when changing the terminal type and specifying the flag bits. The flag bits require further clarification:

TM_SNL Handle new lines specially, if the terminal driver is so equipped.

TM_ANL Provide a carriage return and new line when writing beyond column 80.

TM_LCF Immediately before placing a character in the last column and row, delete the top line, print the character in the last column of the now second-to-last row, and then move the cursor to column one of the new last line. This function is requires for terminals that move the cursor to "bad" places when printing in the last position.

TM_CECHO
Echo the control sequences, such as "cursor up", when received.

TM_CINVIS
Do not pass the cursor control characters to the user program on input.

SEE ALSO

/usr/include/sys/sgtty.h
/usr/include/sys/mx.h
/usr/include/sys/trans.h
stty:o(2), fcntl(2)

ASSEMBLER

(ioctl = 54.)
(filedes in r0)
sys ioctl; request; argp

NAME

kill – send signal to a process

SYNOPSIS

kill (target, sig);

DESCRIPTION

There are several different cases of *kill*:

kill(target,+sig)
Sends the signal *sig* to the process with process id *target* if the sending process's uid matches the uid of *target*. If the sending process's uid is *root*, the signal is sent unconditionally.

kill(target,–sig)
Sends the signal *sig* to all processes in the process group *target* whose uids match the sending process's uid. If the sending process's uid is *root*, all processes in the process group *target* will receive the signal.

kill(0,+sig)
Sends the signal *sig* to all members of the process group of the sending process whose uids match the uid of the sending process. If the sending process's uid is *root*, all members of the sending process's group receive the signal. Note that the sending process will also receive the signal.

kill(0,–sig)
Sends the signal *sig* to all members of the process group of the sending process whose uids match the sending process's uid - except that the sending process will not receive the signal. If the sending process's uid is *root*, all members of the sending process's process group – except the sending process – will receive the signal.

kill(–1,+sig)
Sends the signal *sig* to all processes whose uids match the sending process's uid. If the sending process's uid is *root*, all processes - except 0 and 1 - will receive the signal.

kill(–1,–sig)
Sends the signal *sig* to all processes whose uids match the sending process's uid – except that the sending process will not receive the signal. If the sending process's uid is *root*, all processes – except 0, 1, and the sender – will receive the signal.

kill(target,0)
Reserved for future expansion.

See *signal*(2) for a list of signals.

SEE ALSO

signal(2), kill(1)

DIAGNOSTICS

The error bit (c-bit) is set if the process does not have the same user ID and the user is not super-user, or if the process does not exist.

ASSEMBLER

(kill = 37.; not in assembler)
(process number in r0)
sys kill; sig

NAME

link − link to a file

SYNOPSIS

int link (name1, name2)
char *name1, *name2;

DESCRIPTION

A link to *name1* is created; the link has the name *name2*. Either name may be an arbitrary path name.

SEE ALSO

ln(1), unlink(2)

DIAGNOSTICS

Zero is returned when a link is made; −1 is returned when *name1* cannot be found; when *name2* already exists; when the directory of *name2* cannot be written; when an attempt is made to link to a directory by a user other than the super-user; when an attempt is made to link to a file on another file system; when a file has too many links.

ASSEMBLER

(link = 9.)
sys **link; name1; name2**

NAME

lseek − move read/write pointer

SYNOPSIS

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

DESCRIPTION

The file descriptor refers to a file open for reading or writing. The read (resp. write) pointer for the file is set as follows:

> if *whence* is 0, the pointer is set to *offset* bytes.
>
> if *whence* is 1, the pointer is set to its current location plus *offset.*
>
> if *whence* is 2, the pointer is set to the size of the file plus *offset.*

The returned value is the resulting pointer location.

The obsolete function *tell(fildes)* is identical to *lseek*(*fildes*, 0L, 1).

SEE ALSO

creat(2), open(2), fseek(3S)

DIAGNOSTICS

−1 is returned for an undefined file descriptor, seek on a pipe, or seek to a position before the beginning of file. **SIGSYS** is raised if *whence* is not 0, 1, or 2.

BUGS

Lseek is a no-op on character special files.

ASSEMBLER

(lseek = 19.)
(file descriptor in r0)
sys lseek; offset1; offset2; whence
(new pointer location in r0-r1)

Offset1 and *offset2* are the high and low words of *offset.*

NAME

maus, getmaus, freemaus, enabmaus, dismaus, switmaus − multiple access user space operations

SYNOPSIS

getmaus (name, mode)
char *name;

freemaus (mausdes)

char *enabmaus (mausdes)

dismaus (vaddr)
char *vaddr;

char *switmaus (mausdes, vaddr)
char *vaddr;

DESCRIPTION

MAUS is a dedicated portion of core memory, which may be subdivided in, logical subsections. See *maus*(4) for a discussion of MAUS layout. These subsections are referenced via entries in the UNIX file system which may be used as arguments to the UNIX *open* system call or the *getmaus* system call. Opening such a special file results in a file descriptor being returned which may be subsequently used with other file system calls like *read*, *write*, *seek* and *close* in the standard manner.

Performing the *getmaus* primitive on a maus special file returns a maus descriptor which is analogous to a file descriptor in many ways. This maus descriptor may be subsequently used with the *enabmaus* primitive to attach the described maus subsection to the user's address space. If the *enabmaus* primitive is used repetitively on the same maus descriptor different virtual addresses will be returned on each call until all memory mapping registers have been used; at which time an error is returned. Note that every active instance of maus requires the allocation of a separate memory mapping register since no register may point to more than one maus segment at a time.

Once *enabmaus* has been used the *dismaus* primitive may be utilized to remove active instances of maus from the user's address space. (In reality, *enabmaus* and *dismaus* are special cases of the *switmaus* primitive described below.)

Finally, *freemaus*, deallocates a maus descriptor so that it may be reassigned by *getmaus*. Note that if a maus descriptor has been enabled it may still be freed: the virtual address returned by *enabmaus* remains in the user's address space until a *dismaus* primitive is utilized on the virtual address in question.

The maus primitives are defined as follows:

if *function* is a **0**, **1**, or **2** (*getmaus*(*name,mode*) from C), the maus file described by *argy* (*name* from C) is accessed to determine if the read, write, or read/write permission as specified by *function* (*mode* from C) should be granted to the specified user. This permission check is in accordance with the standard UNIX file protection. The file specified must be a special maus file. This primitive returns a maus descriptor which must be saved for future use with *freemaus* and *enabmaus*. This primitive is similar to the open system call in many respects.

if *function* is **3** (*freemaus*(*mausdes*) from C), the maus descriptor described by *argy* (*mausfes* from C) is deallocated from the process. Any further attempts to use the value as a maus descriptor will result in an error being returned.

if *function* is **4** (*switmaus*(*mausdes*,*vaddr*) from C), the system will select the user data memory mapping register specified by *argy* (*vaddr* from C), and load it so that the maus segment specified by *argx* (*mausdes* from C), becomes part of the user's virtual address space. When using the C interface, the value returned by *switmaus* is the old maus descriptor associated with *vaddr*; if *vaddr* had not been associated with a maus descriptor, −2 is returned. For the assembly interface, the value returned is a pointer to the start of this maus area which may be used like any assembly pointer, but should be preserved for future maus system calls. If *argx* is a −1, (*dismaus*(*vaddr*) from C), the specified virtual address is removed from the user's address space. The C interface returns the maus descriptor which had been associated with *vaddr*; if *vaddr* had not been enabled then -2 is returned. The assembly interface returns some value not equal to −1 (unless there has been an error). If *argy* is −1 (*enabmaus*(*mdes*) from C), the first available memory mapping register is allocated and used. If both arguments are −1, an error will be returned only if there are no unused user memory mapping registers. An error indication is always returned if no memory mapping registers are available or if an address is specified which is in use for program text, data or stack. When expecting a maus descriptor to be returned, for example after a *dismaus (vaddr)*, a -2 return means that no maus descriptor had been enabled with the virtual address given. In all cases, a −1 return means error.

FILES

/dev/maus/*

RULES OF THE ROAD

1) Maus descriptors are inherited across forks and executes. Note that if the new process executed has text or data which wants to occupy the memory currently open to maus, the execute will fail.

2) Maus virtual addresses are inherited across forks.

3) If the *break* system call is used to increase the user's size to the point where an additional memory mapping register is needed and maus is utilizing the next contiguous memory mapping register, the *break* will fail. The user may then utilize *enabmaus* and *dismaus* to reassign the maus virtual address(es). This can be done by doing successive *enabmaus* system calls until the desired virtual address is reached and then disabling the unneeded addresses before using the *break* system call. Alternatively, the user could disable all the active maus segments, use the *break* system call, and then reenable the maus segments.

4) Since the memory mapping hardware does not allow a write-only segment, when the user requests write-only maus via the *getmaus* primitive he is actually granted read-write permission assuming the file system protection tests pass. Only write permission of the maus special file is tested in this case.

SEE ALSO

break(2), open(2), maus(4)

DIAGNOSTICS

From assembler the error bit is set for any error. From C, a −1 return indicates an error.

ASSEMBLER

(maus = 58.; not in assembler)
(function in R0)
(argx in R1)
sys maus; argy

NAME

mdate − set modified date on file

SYNOPSIS

```
mdate (file, time)
char *file;
int time[2];
```

DESCRIPTION

File is the address of a null-terminated string naming a file; the modified time of the file is set to the time given in registers r0 and r1 (resp. in the vector which is the second argument). See *time*(2) for the units and epoch.

This call is allowed only to the super-user or to the owner of the file.

Mdate is obsolete − use *utime*(2) instead.

SEE ALSO

time(2), utime(2)

DIAGNOSTICS

Error bit is set if the user is neither the owner nor the super-user or if the file cannot be found. From C, a negative return indicates an error, a 0 return indicates success.

BUGS

Caution: setting back the date of a file probably will prevent it from being dumped by an incremental dump.

NAME

menab, mdisab, msend, mrecv, mctl — send and receive messages

SYNOPSIS

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

menab (name, flags)
short name;
short flags;

mdisab (disp)
short disp;

msend (&mstr, buf, size)
mrecv (&mstr, buf, size)
struct mstr mstr;
caddr_t buf;
short size;

mctl (&mstr, command, arg, size)
struct mstr mstr;
short command;
caddr_t arg;
short size;
```

DESCRIPTION

Messages are a very fast form of interprocess communication. Messages are stored on named queues. A process may send a message to any queue for which it has permission. A process can attach to one and only one queue at a time to receive messages according to the permissions associated with the queue. (There may, however, be synonyms for the same queue, see below.)

menab(name,flags)

Enable message reception via the queue *name*. If the queue does not already exist, create it, giving it the characteristics specified by *flags*. If the queue already exists, attempt to attach the existing queue. Attaching an existing queue will succeed only if the following conditions are met:

1) The *flags* argument does match the permissions for the queue (see <**sys/msg.h**>.)

2) The **MXCLUDE** bit is not set for the queue. (This bit is always cleared by the system when the last process disconnects from a queue, hence it is always possible for a process with the proper permissions to attach a queue if no one else is attached.)

3) The **MOTHR** and **MGRPR** permissions in combination with the queue's and process' user and group ids allow the attempt. These permissions are interpreted in the same way as the normal UNIX file permissions: see *access*(2).

The *flags* are as follows:

MNODESTROY — Do not destroy the queue when the last process detaches. This is the default action. When either MNODESTROY or MDESTROY is specified by *menab*() it is used if the process dies or exits without specifically detaching the queue with a *mdisab()*.

MDESTROY — Destroy the queue when the last process detaches. All messages remaining on the queue at the time of destruction, which require acknowledgement (the MACKREQ flag was set when they were sent), are returned to the sending process if possible, with a type of MACKTYP.

MXCLUDE Do not allow any other process to attach to this queue. This remains in force as long as the current process is attached.

MPRIQ Queue messages in order of priority based on *ms_type*. Normally messages are queued in order of arrival, first-in, first-out (FIFO). In a priority queue, messages with larger *ms_stype*'s are stored before messages with lower *ms_stype*'s. (See *mrecv* below)

MGRPR Allow any process with the same group id as the group id of the creating process to read the queue, i.e. attach the queue for receiving.

MGRPW Allow any process with the same group id as the group id of the creating process to write the queue, i.e. send messages to the queue.

MOTHR Allow any process whose user id and group id are different from the creating process' ids to read the queue, i.e. attach the queue for receiving.

MOTHW Allow any process whose user id and group id are different from the creating process' ids to write the queue, i.e. send messages to the queue.

Upon a successfully attaching to a queue, *menab*() returns the number of processes attached to the queue.

mdisab(disp)

Disable message reception and detach the queue. *disp* contains either the **MNODESTROY** or the **MDESTROY** flag, stating what the disposition of the queue is to be if this is the last process releasing the queue. This overrides the disposition specified during the *menab()*.

msend(mstr,buf,size)

Send a message contained in *buf*, which is of *size* bytes to the queue specified by the *mstr* structure. *mstr* should contain the queue name and the system name to which the message is to be sent (in *ms_qname* and *ms_system*). It should also contain the message subtype in *ms_stype* and the message type and flags, specified in *ms_flags*. Message subtypes can take any value from *1* to *127*.

The flags and types are as follows:

MNOBLOCK Do not wait if the message cannot be sent (or received for *mrecv*) immediately, but return with an appropriate error message.

MNOCOPY Do not copy the message out of the user space. Instead adjust the memory mapping so that it is no longer apart of the user's address space. For this feature to work the system must have the feature enabled and the message itself must be in a section of shared memory. Initially shared memory for messages may be gotten using *smget* (see *shmem*(2)). During an *msend*(), if the address of the buffer supplied is not shared memory and the **MNOCOPY** flag is set, then the *msend*() will fail. Messages sent *without* the **MNOCOPY** flag cannot be larger than **MAXMLEN**. Messages sent as **MNOCOPY** are limited only by the amount of shared memory that can be in existence at one time, a system definable parameter. When a process receives a **MNOCOPY** message, the shared memory message space is mapped into the address space of the receiver and *ms_addr* is set to point to the beginning of this shared memory segment. The **MNOCOPY** flag will be on in *ms_flags*. Messages received with the **MNOCOPY** flag set may be sent to other processes with it set or the shared memory space may be returned to the operating system using *smfree* (see *shmem*(2)). If a process tries to receive a **MNOCOPY** message and it cannot be mapped into the user's address space, as much as possible is copied into the user supplied buffer and the **MNOCOPY** flag is turned off.

MACKREQ An acknowledgement is required for this message. If a message with this type is still on a queue when it is destroyed, the operating system will change its type to **MACKTYP** and attempt to return it to the sender.

MDATATYP Declares that this message is a data type message. This type has no meaning to the operating system and is supplied to be used by users.

MCTLTYP Declares that this message is a control type message. This type has no meaning to the operating system and is supplied to be used by users.

MINTRTYP Declares that this message is an interrupt type message. This type has no meaning to the operating system and is supplied to be used by users.

MACKTYP Declares that this message is an acknowledgement. The operating system will not allow a message to be sent which has **MACKREQ** set and is of type **MACKTYP**. The operating system will change the type of any message being returned to sender to **MACKTYP**. (See **MACKREQ** above.)

Upon successfully sending a message, *msend*() returns the number of bytes of message actually sent.

mrecv(mstr,buf,size)

Receive a message. Normally the message will be placed in *buf*, and truncated to *size* bytes if the message is bigger than the buffer. Messages received with the **MNOCOPY** flag on will not use *buf*. *mstr* should initially contain the subtype (*ms_stype*) and optionally the **MNOBLOCK** flag, if waiting is not desired. The remainder of *mstr* will be filled in by the operating system dependent upon the message actually being received. *ms_qname* and *ms_system* will contain the name of the queue to which the sending process is attached. If the message sender does not have messages enabled, then *ms_qname* will be 0. *ms_rqname* will contain the name of the queue that the message was actually sent to. (See **MAPQ** below.) The subtype and the type of the queue (FIFO or priority) determine which message will be received.

FIFO

ms_type = 0

Return next message of any subtype. The subtype of the message actually received will be placed by the operating system into *mstr*.

ms_type = 1−127

Return only a message of this specific type. If the message queue is full and there isn't a message of the specific type on the queue and someone attempts to send a message of the desired type, the message will be sent and the receiver will wake up. This will not work if there are multiple receivers sleeping on different non-zero types. In this case one of the processes may never wakeup. Receiving a specific message type from a FIFO message queue should be used very carefully.

Priority

ms_type = 0−127
Return the first message whose subtype is greater than or equal to *ms_stype* in the receiver's *mstr*.

mctl(mstr,command,arg,size)
Fetch and change various parameters for queues. The commands are:

GETMSTAT Returns an *mstats* structure containing the number of messages presently on the queue, the maximum number allowable, the owner and group of the queue, the number of processes attached to the queue, and the modes and disposition of the queue.

SETMQLEN Sets the maximum number of messages that a queue can contain to *command.ms_smqlen*. This number cannot be greater than **MAXMSGL** (See **<sys/param.h>**). Only processes with the same user id as the queue or which are super−user can change the maximum queue length.

SETREMQ This allows one queue to be declared as the *remote* queue. All messages destined for systems other than the present system are routed to this queue. The process reading the remote message queue is responsible for actually getting the message to the remote system by whatever means it is programmed to use. *ms_system*, *ms_qname*, and *ms_rqname* have special meanings when a remote queue manager receives and sends messages. When receiving messages *ms_qname* contains the name of a local queue attached to the sending process; *ms_system* continues to contain the name of the remote system to which the message is to be sent; and *ms_rqname* contains the name of the remote system queue to which the message is to be sent. When the process attached to the remote message queue sends a message *ms_qname* always specifies a local queue name. The operating system takes the values of *ms_system* and *ms_rqname* and places them into *ms_system* and *ms_qname* of the final message so that the local receiver of the message sees the message as having arrived from that system and remote queue.

SETSPYQ This is a debugging aid. It specifies that a copy of all messages sent to the queue specified by *mstr* be sent to the queue *arg.ms_spyq*. There can only be one spy queue in the system at a time.

MAPQ This command allows the creation and removal of synonym queue names. A message sent to synonym queue name is sent to the real queue, but with *ms_rqname* set to the synonym queue name to which the message was directed. In this way the receiving process will know where the sender thought the message was going. Note that the synonym queue has all the permissions of the original queue and that the synonym will disappear when the original queue is destroyed. It is illegal to create a synonym which is the same as the original and it is also illegal to attach to a synonym queue. To create or remove a synonym queue the process performing the **MAPQ** function must have read permission for the real queue. To create a synonym, *mstr* specifies a real queue and arg.ms_synq is the synonym queue name to be associated with the real queue. If *mstr.ms_qname* is 0 and *arg.ms_synq* specifies a current synonym queue name, then the synonym queue name is removed.

Messages reception remains enabled across *exec*, but not across *fork*.

In creating queue names the following convention is recommended. All system wide permenant queue names should be defined in the header file, **/usr/include/msgqueues.h**. All such permenant queue names should be negative numbers (0100000 to 0177777), thereby leaving the positive numbers available to processes which need a temporary queue for acknowledgements or which are using the old message veneer. (See *msg*(3)). Such processes may therefore create temporary queues with names equal to their *pid* and be assured that these names will not collide with permenant queue names since *pids* are never negative.

The format of <**sys/msg.h**> is as follows:

```
/*              @(#)msg.h       3.1             */
/*
 * Message Control Structures
 */

typedef         short           queue_t;


/*
 * Modes for menab and mdisab. (ST.mq_modes)
 * For mdisab only the MDESTROY flag is meaningful.
 */
#define MNODESTROY      0000            /* Retain queue when unreferenced */
#define MDESTROY        0001            /* Destroy queue when unreferenced */
#define MOTHR           0002            /* Other read permission */
#define MOTHW                           0004            /* Other write permission */
#define MXCLUDE                         0010            /* Only one process may attach */
#define MGRPR           0020            /* Group read permission */
#define MGRPW                           0040            /* Group write permission */
#define MPRIQ           0100            /* Priority type queue */

#define MDEFAULT        (MNODESTROY|MOTHR|MOTHW|MXCLUDE|MGRPR|MGRPW)


/*
 * commands for mctl call
 */
#define GETMSTAT        0               /* get message status */
#define SETMQLEN        1               /* set message queue length */
#define SETREMQ                         2               /* set remote message queue */
#define SETSPYQ                         3               /* set spy parameters */
#define MAPQ            4               /* create/destroy synonym queues */


/*
 * structure of arg for GETMSTAT command of mctl
 */
struct mstats {
                short           mq_cnt;                         /* number in queue */
                short           mq_mslim;       /* maximum queue size */
                short           mq_uid;                         /* "owner" uid */
                short           mq_gid;                         /* "owner" gid */
                char            mq_refc;        /* no. attached to queue */
                char            mq_modes;       /* permissions and disposition */
};

/*
```

```
 * structure of arg for SETMQLEN command
 */
struct sctmq {
                short           ms_smqlen;      /* maximum queue length */
};

/*
 * For the SETREMQ command the arg and size arguments to
 * mctl are not used. The queue name specified in the first
 * argument to mctl is the queue which becomes the remote queue.
 * If this queue name is zero, the current remote queue is
 * disconnected.
 */

/*
 * structure of arg for SETSPYQ command
 * The first arg to mctl specifies the queue to be spied upon.
 * This arg specifies the queue to which a copy of the data is
 * to be sent.
 */
struct setspyq {
                queue_t         ms_spyq;
};

/*
 * structure of arg for the MAPQ command
 * The first arg to mctl specifies the existing queue
 * to which the synonym is to be mapped. If it specifies a
 * qname of zero any existing synonym with the name
 * specified in the synq structure is eliminated.
 * To successfully create or remove a queue synonym the
 * user doing the MAPQ command must have read permission
 * for the real queue.
 */
struct synq {
                queue_t         ms_synq;
};

/*
 * structure for sending and receiving messages
 */
struct mstr {
                long            ms_system;      /* system name  */
                queue_t         ms_qname;       /* queue name   */
                char            ms_stype;       /* message sub-type/priority */
                char            ms_flags;
                caddr_t         ms_addr;        /* address for mrecv */
                queue_t         ms_rqname;      /* queue msg was sent to */
                short           ms_uid;                         /* sender's user id */
                short           ms_gid;                         /* sender's group id */
};

/*
 * Flag values for ms_flags
 */
#define MNOBLOCK                001             /* Non-blocking send and recv */
#define MNOCOPY                                 002             /* Remap segment-no copy if possible */
#define MACKREQ                                 004             /* Ack required */
#define MDATATYP                000             /* Data message */
#define MCTLTYP                                 010             /* Control message */
#define MINTRTYP                020             /* Interrupt message */
#define MACKTYP                                 030             /* Ack message */
#define MTYPMSK                                 030             /* Mask of type bits */
```

```
#ifdef KERNEL

#define MFLGCARE                (MOTHR|MOTHW|MGRPR|MGRPW|MPRIQ)

#define PMSG    PZERO+5                         /* message sleep priority */
#define MSGIN  B_WRITE
#define MSGOUT                  B_READ
#define MREAD02
#define MWRITE                  04

#define MDISAB                  0
#define MENAB 1
#define MSEND 2
#define MRECV 3
#define MSGCTL                  4

#define NORMAL_SEND             00000           /* Normal msg - user to user */
#define REM_USR                                 00400           /* Remote msg - daemon to user */
#define REM_SEND                01000           /* Remote msg - user to daemon */

/*
 * State bits
 */
#define IP_QWANT                0100            /* msg queue wanted */
#define IP_WANTED               0200            /* resource is desired */

struct msghdr {
                struct msghdr *mq_forw;
                union {
                                struct {
                                                short           mq_size;
                                                queue_t         mq_sender;
                                                long            mq_system;
                                                paddr_t         mq_addr;
                                                queue_t         mq_rqname; /* remote queue name */
                                                char            mq_stype;
                                                char            mq_flags;
                                                short           mq_muid;
                                                short           mq_mgid;
                                }ms;
                                struct {
                                                struct msghdr *mq_last;
                                                queue_t mq_name;
                                                char            mq_twant;       /* Wanted for type */
                                                char            mq_state;
                                                struct mstats st;
                                }qu;
                }UN;
};

/*
 * Shorthand notations for accessing elements of above structure
 */

#define QU      UN.qu
#define MS      UN.ms
#define ST      UN.qu.st

/*
 * Message related measurements
 */
struct M_MEAS {
                short           qinuse;                         /* number of queues in use */
```

```
                 short      qtblovr;      /* no. of queue table overflows */
                 short      mtblovr;      /* no. of msg table overflows */
                 long       msgsent;      /* no. of msgs sent */
                 long       msgrecv;      /* no. of msgs received */
                 long       msgflsh;      /* no. of msgs flushed */
     };
     #endif
```

DIAGNOSTICS

A −1 is returned for any one of a number of error conditions. An error occurs when enabling messages if no queue can be allocated or if the process is attempting to connect to a queue that does not have the appropriate permissions; it is also erroneous to attempt to disable message reception if it is not enabled. When trying to send messages, errors occur because the message is too long, the specified message queue or system does not exist, the type or priority specified is not valid, the **MNOCOPY** bit is used incorrectly, or, for conditional sends, the system message buffers are temporarily full or the receiver has an excessive number of messages on its queue. When receiving messages, errors may occur because the process has not enabled message reception, the requested priority is invalid, or, for conditional receives, a message of the requested type is not on the queue. It is also illegal to set the message limit (via *mctl*) to a value larger than defined by **MAXMSGL** or to specify a *mctl* for a queue that the user could not connect to.

FILES

/usr/include/sys/param.h
/usr/include/sys/msg.h

BUGS

It may not be possible to return errors correctly when trying to send messages to remote systems.

SEE ALSO

access(2), shmem(2), msg(3)

NAME

mknod − make a directory or a special file

SYNOPSIS

```
int mknod (name, mode, addr)
char *name;
int mode, addr;
```

DESCRIPTION

Mknod creates a new file whose name is the null-terminated string pointed to by *name*. The mode of the new file (including directory and special file bits) is initialized from *mode*. (The protection part of the mode is modified by the process's mode mask; see *umask*(2)). The first block pointer of the i-node is initialized from *addr*. For ordinary files and directories, *addr* is normally zero. In the case of a special file, *addr* specifies which special file.

Mknod may be invoked only by the super-user.

SEE ALSO

mkdir(1), mknod(1), fs(5)

DIAGNOSTICS

Zero is returned if the file has been made; a −1 if the file already exists or if the user is not the super-user.

ASSEMBLER

(mknod = 14.)
sys mknod; name; mode; addr

NAME

mount, umount − mount or remove file system

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mount.h>
int mount (special, name, mtflags)
char *special, *name;
int mtflags;
```

DESCRIPTION

Mount announces to the system that a removable file system, *special*, is now mounted on the innode associated with *name*. From now on, references to file *name* will refer to the root file on the newly mounted file system. *Special* and *name* are pointers to null-terminated strings containing the appropriate path names.

Name must exist already. *Name* must be a directory (unless the root of the mounted file system is not a directory). Its old contents are inaccessible while the file system is mounted.

The *mtflags* argument passes two mount flags to the operating system. **M_RONLY** says that the file system is to be read-only. Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted. **M_NOSETUG** says that the set user/group feature of the *exec* system call is to be disabled for all executions taking place from this file system. **M_NOCBO** says that opens of character and block special devices will not be allowed from this file system.

Mount may be issued only by the super-user.

SEE ALSO

mount(1), umount(2)

DIAGNOSTICS

Mount returns 0 if the action occurred; −1 if *special* is inaccessible or not an appropriate file; if *name* does not exist; if *special* is already mounted; if *name* is in use; or if there are already too many file systems mounted.

ASSEMBLER

```
(mount = 21.)
sys mount; special; name; rwflag
```