CTIX™ OPERATING SYSTEM MANUAL

Version C
Volume 3

This manual was prepared on a Convergent Technologies S/640 Computer System and was printed on an Apple LaserWriter II Laser Printer.

Second Edition (November 1989) 09-02264-01
Update Notice 1 (November 1990) 09-02578

TABLE OF CONTENTS: VOLUME 3

3. Subroutines and Libraries

NAME

fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky - IEEE floating-point environment control

SYNOPSIS

```
#include <ieeefp.h>

typedef enum {
        FP_RN=0,        /* round to nearest */
        FP_RZ=0x10,     /* round to zero (truncate) */
        FP_RM=0x20,     /* round to minus */
        FP_RP=0x30,     /* round to plus */
        } fp_rnd;

fp_rnd fpgetround( );

fp_rnd fpsetround(rnd_dir)
fp_rnd rnd_dir;

#define  fp_except   int
#define  FP_X_INV    0x80  /* invalid operation */
                           /* exception */
#define  FP_X_OFL    0x40  /* overflow */
                           /* exception */
#define  FP_X_UFL    0x20  /* underflow */
                           /* exception */
#define  FP_X_DZ     0x10  /* divide-by-zero */
                           /* exception */
#define  FP_X_IMP    0x08  /* imprecise (loss */
                           /* of precision) */

fp_except fpgetmask( );

fp_except fpsetmask(mask);
fp_except mask;

fp_except fpgetsticky( );

fp_except fpsetsticky(sticky);
fp_except sticky;
```

DESCRIPTION

These routines let the user change the behavior on the occurrence of any of five floating-point exceptions: divide-by-zero, overflow, underflow, imprecise (inexact) result, and invalid operation. The routines also change the rounding mode for floating-point operations. When a floating-point exception occurs, the corresponding sticky bit is set (1), and if the mask bit is enabled (1), the trap takes place. The routines are valid only on systems that are equipped with floating-point accelerator hardware; otherwise, floating-point operations are compiled differently and handled in software.

The *fpgetround*() routine returns the current rounding mode.

The *fpsetround*() routine sets the rounding mode and returns the previous rounding mode.

The *fpgetmask*() routine returns the current exception masks.

The *fpsetmask*() routine sets the exception masks and returns the previous setting.

The *fpgetsticky*() routine returns the current exception sticky flags.

The *fpsetsticky*() routine sets (clears) the exception sticky flags and returns the previous setting.

The environment for Convergent computers with either a MC68040 CPU or a combined MC68020 CPU with MC68881 or MC68882 floating-point processor follows:

- Rounding mode set to nearest(FP_RN)
- Divide-by-zero
- Floating-point overflow
- Invalid operation traps enabled

SEE ALSO

isnan(3C).

CAVEATS

The utilities described in this man page are applicable only for computers that are equipped with either the MC68040 microprocessor, or both the MC68020 microprocessor CPU and the MC68881, or the MC68882 microprocessor for a hardware-floating point accelerator. Programs that invoke these utilities are run on computers without the floating-point hardware and result in no operation and no returned error message for the particular function.

One must clear the sticky bit to recover from the trap and to proceed. If the sticky bit is not cleared before the next trap occurs, a wrong exception type may be signaled.

For the same reason, when calling *fpsetmask*( ), the user should make sure that the sticky bit corresponding to the exception being enabled is cleared.

WARNINGS

The *fpsetsticky*() routine modifies all sticky flags; *fpsetmask*() changes all mask bits.

C requires truncation (round to zero) for floating point to integral conversions. The current rounding mode has no effect on these conversions.

NAME

getspent, getspnam, setspent, endspent, fgetspent, lckpwdf, ulckpwdf - get shadow password file entry

SYNOPSIS

#include <shadow.h>

struct spwd *getspent ()

struct spwd *getspnam (name)
char *name;

int lckpwdf ()
int ulckpwdf ()

void setspent ()

void endspent ()

struct spwd *fgetspent (fp)
FILE *fp;

DESCRIPTION

The *getspent* and *getspnam* routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/shadow** file. Each line in the file contains a shadow password structure (**spwd**), declared in the < **shadow.h** > header file:

```
struct spwd{
        char    *sp_namp;
        char    *sp_pwdp;
        long    sp_lstchg;
        long    sp_min;
        long    sp_max;
};
```

The *getspent* routine, when first called, returns a pointer to the first **spwd** structure in the file; thereafter, it returns a pointer to the next **spwd** structure in the file. This way, successive calls can be used to search the entire file. The *getspnam* routine searches from the beginning of the file until a login matching *name* is found, and then returns a pointer to the particular structure in which it was found. The *getspent* and *getspnam* routines populate the **sp_min** or **sp_max** field with -1 if the corresponding field in **/etc/shadow** is empty. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

The **/etc/.pwd.lock** file is the lock file, which is used to coordinate modification access to the password files in **/etc/passwd** and **/etc/shadow**. The *lckpwdf*()

and *ulckpwdf*( ) routines are used to gain modification access to the password files, through the lock file. A process first uses *lckpwdf*( ) to lock the lock file, thereby gaining exclusive rights to modify the **/etc/passwd** or **/etc/shadow** file. Upon completing modifications, a process should release the lock on the lock file by using *ulckpwdf*( ). This lock mechanism prevents simultaneous modification of the password files.

The *lckpwdf*( ) routine attempts to lock the file **/etc/.pwd.lock**. If the file is already locked, *lckpwdf*( ) tries for 15 seconds to lock the file. If unsuccessful, *lckpwdf*( ) returns a -1; if successful within 15 seconds, *lckpwdf*( ) returns a return code other than -1.

The *ulckpwdf*( ) routine attempts to unlock the file **/etc/.pwd.lock**. If successful, *ulckpwdf*( ) returns a 0; if unsuccessful (if the file is not locked), *ulckpwdf*( ) returns a -1.

A call to the *setspent* routine has the effect of rewinding the shadow password file to allow repeated searches. The *endspent* routine may be called to close the shadow password file when processing is complete.

The *fgetspent* routine returns a pointer to the next **spwd** structure in the stream *fp*, which matches the format of **/etc/shadow**.

FILES

/etc/shadow
/etc/passwd
/etc/.pwd.lock

SEE ALSO

putspent(3X).

DIAGNOSTICS

A NULL pointer is returned on **EOF** or error.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

WARNING

If a program not otherwise using standard I/O uses this routine, the size of the program increases more than might be expected.

This routine is for internal use only; compatibility is not guaranteed.

NAME

monitor - prepare execution profile

SYNOPSIS

#include <mon.h>

void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
WORD *buffer;
int bufsize, nfunc;

DESCRIPTION

An executable program created by **cc-p,** it automatically includes calls for *monitor* with default parameters; *monitor* need not be called explicitly.

monitor is an interface to *profil*(2). *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user-supplied) array of *bufsize* WORDs (defined in the *<mon.h>* header file). *monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *lowpc* may not equal 0 for this use of *monitor*. At most, *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **-p** of *cc*(1) are recorded.

prof(1) can then be used to examine the results.

The name of the file written by *monitor* is controlled by the environment variable PROFDIR. If PROFDIR does not exist, **mon.out** is created in the current directory. If PROFDIR exists but has no value, *monitor* does not do any profiling and creates no output file. Otherwise, the value of PROFDIR is used as the name of the directory in which to create the output file. If PROFDIR is *dirname*, then the file written is *dirname/pid*.**mon.out**, where *pid* is the program's process ID. (When *monitor* is called automatically by compiling via **cc -p**, the file created is *dirname/pid.progname*, where *progname* is the name of the program.)

The following discussion is a sketch of *monitor* usage.

For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no less than one half of the range of locations sampled.

To profile the entire program, put the following at the start of **main()** :

```
extern etext;
...
monitor ((int (*)())2, &etext, buf, bufsize, nfunc);
```

etext lies just above all the program text; see *end*(3C).

To stop execution monitoring and write the results, put the following at the end of **main()** :

```
monitor ((int (*)())0, 0, 0, 0, 0);
```

Do not compile with the **-p** option. Run the program and use *prof(1)* to view the results in the output file **mon.out** .

FILES

mon.out

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

BUGS

The ``*dirname/pid*.mon.out'' form does not work; the ``*dirname/pid.progname*'' form (automatically called via **cc -p**) does work.

NAME

sleep - suspend execution for interval

SYNOPSIS

unsigned sleep (seconds)
unsigned seconds;

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) because scheduled wakeups occur at fixed 1-second intervals (on the second, according to an internal clock), and (2) because any caught signal terminates the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the ''unslept'' amount (the requested time minus the time actually slept), in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the *sleep* routine returns. But if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2), pause(2), signal(2).

WARNING

sleep uses *signal*(2), not *sigset*(2), to reset the caller's **SIGALRM** handler routine. Therefore, the signal action is reset to its default action on execution of the **SIGALRM** handler. This is probably not what the programmer intended if *sigset*(2) had originally been used to set the signal action.

sleep uses a longjmp, which returns to the *sleep* context when the *alarm*(2) signal handler routine is executed. This may cause premature preemption and loss of context from other nested signal handler routines.