NAME

decimal_to_single, decimal_to_double, decimal_to_extended – convert decimal record to floating-point value

SYNOPSIS

```
#include <floatingpoint.h>

void decimal_to_single(px, pm, pd, ps)
single *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;

void decimal_to_double(px, pm, pd, ps)
double *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;

void decimal_to_extended(px, pm, pd, ps)
extended *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;
```

DESCRIPTION

The **decimal_to_floating()** functions convert the decimal record at **pd* into a floating-point value at **px*, observing the modes specified in **pm* and setting exceptions in **ps*. If there are no IEEE exceptions, **ps* will be zero.

pd->sign and *pd->fpclass* are always taken into account. *pd->exponent* and *pd->ds* are used when *pd->fpclass* is *fp_normal* or *fp_subnormal*. In these cases *pd->ds* must contain one or more ascii digits followed by a NULL. **px* is set to a correctly rounded approximation to

(pd->sign)*(pd->ds)*10(pd->exponent)**

Thus if *pd->exponent* == –2 and *pd->ds* == "1234", **px* will get 12.34 rounded to storage precision. *pd->ds* cannot have more than **DECIMAL_STRING_LENGTH-1** significant digits because one character is used to terminate the string with a NULL. If *pd->more != 0* on input then additional nonzero digits follow those in *pd->ds; fp_inexact* is set accordingly on output in **ps*.

**px* is correctly rounded according to the IEEE rounding modes in *pm->rd*. **ps* is set to contain *fp_inexact*, *fp_underflow*, or *fp_overflow* if any of these arise.

pd->ndigits, *pm->df*, and *pm->ndigits* are not used.

strtod(3), **scanf**(3), **fscanf**(3), and **sscanf**(3) all use **decimal_to_double**.

SEE ALSO

scanf(3S), **scanf**(3V), **strtod**(3)

NAME

econvert, fconvert, gconvert, seconvert, sfconvert, sgconvert, ecvt, fcvt, gcvt – output conversion

SYNOPSIS

```
#include <floatingpoint.h>

char *econvert(value, ndigit, decpt, sign, buf)
double value;
int ndigit, *decpt, *sign;
char *buf;

char *fconvert(value, ndigit, decpt, sign, buf)
double value;
int ndigit, *decpt, *sign;
char *buf;

char *gconvert(value, ndigit, trailing, buf)
double value;
int ndigit;
int trailing;
char *buf;

char *seconvert(value, ndigit, decpt, sign, buf)
single *value;
int ndigit, *decpt, *sign;
char *buf;

char *sfconvert(value, ndigit, decpt, sign, buf)
single *value;
int ndigit, *decpt, *sign;
char *buf;

char *sgconvert(value, ndigit, trailing, buf)
single *value;
int ndigit;
int trailing;
char *buf;

char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

DESCRIPTION

econvert( ) converts the *value* to a NULL-terminated string of *ndigit* ASCII digits in *buf* and returns a pointer to *buf*. *buf* should contain at least *ndigit+1* characters. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt*. Thus *buf* == "314" and **decpt* == 1 corresponds to the numerical value 3.14, while *buf* == "314" and **decpt* == −1 corresponds to the numerical value .0314. If the sign of the result is negative, the word pointed to by *sign* is nonzero; otherwise it is zero. The least significant digit is rounded.

fconvert is identical to **econvert**, except that the correct digit has been rounded for Fortran F-format output with *ndigit* digits to the right of the decimal point. *ndigit* can be negative to indicate rounding to the left of the decimal point. The return value is a pointer to *buf*. *buf* should contain at least *310+max(0,ndigit)* characters to accomodate any double-precision *value*.

gconvert( ) converts the *value* to a NULL-terminated ASCII string in *buf* and returns a pointer to *buf*. It produces *ndigit* significant digits in fixed-decimal format, like Fortran F, if possible, and otherwise in floating-decimal format, like Fortran E; in either case *buf* is ready for printing, with sign and exponent. The result corresponds to that obtained by

(void) sprintf(buf,"%gw.n",value) ;

If *trailing*= 0, trailing zeros and a trailing point are suppressed. If *trailing*!= 0, trailing zeros and a trailing point are retained.

seconvert, sfconvert, and **sgconvert( )** are single-precision versions of these functions, and are more efficient than the corresponding double-precision versions. A pointer rather than the value itself is passed to avoid C's usual conversion of single-precision arguments to double.

ecvt( ) and **fcvt( )** are obsolete versions of **econvert( )** and **fconvert( )** that create a string in a static data area, overwritten by each call, and return values that point to that static data. These functions are therefore not reentrant.

gcvt( ) is an obsolete version of **gconvert( )** that always suppresses trailing zeros and point.

IEEE Infinities and NaNs are treated similarly by these functions. "NaN" is returned for NaN, and "Inf" or "Infinity" for Infinity. The longer form is produced when *ndigit* >= 8.

SEE ALSO

printf(3S)

NAME

single_to_decimal, double_to_decimal, extended_to_decimal – convert floating-point value to decimal record

SYNOPSIS

```
#include <floatingpoint.h>

void single_to_decimal(px, pm, pd, ps)
single *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;

void double_to_decimal(px, pm, pd, ps)
double *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;

void extended_to_decimal(px, pm, pd, ps)
extended *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;
```

DESCRIPTION

The **floating_to_decimal()** functions convert the floating-point value at **px* into a decimal record at **pd*, observing the modes specified in **pm* and setting exceptions in **ps*. If there are no IEEE exceptions, **ps* will be zero.

If **px* is zero, infinity, or NaN, then only *pd->sign* and *pd->fpclass* are set. Otherwise *pd->exponent* and *pd->ds* are also set so that

(pd->sign)*(pd->ds)*10(pd->exponent)**

is a correctly rounded approximation to **px*. *pd->ds* has at least one and no more than DECIMAL_STRING_LENGTH–1 significant digits because one character is used to terminate the string with a NULL.

pd->ds is correctly rounded according to the IEEE rounding modes in *pm->rd*. **ps* has *fp_inexact* set if the result was inexact, and has *fp_overflow* set if the string result does not fit in *pd->ds* because of the limitation DECIMAL_STRING_LENGTH.

If *pm->df == floating_form*, then *pd->ds* always contains *pm->ndigits* significant digits. Thus if **px* == 12.34 and *pm->ndigits* == 8, then *pd->ds* will contain 12340000 and *pd->exponent* will contain –6.

If *pm->df == fixed_form* and *pm->ndigits* >= 0, then *pd->ds* always contains *pm->ndigits* after the point and as many digits as necessary before the point. Since the latter is not known in advance, the total number of digits required is returned in *pd->ndigits*; if that number >= DECIMAL_STRING_LENGTH, then *ds* is undefined. *pd->exponent* always gets *–pm->ndigits*. Thus if **px* == 12.34 and *pm->ndigits* == 1, then *pd->ds* gets 123, *pd->exponent* gets –1, and *pd->ndigits* gets 3.

If *pm->df == fixed_form* and *pm->ndigits* < 0, then *pm->ds* always contains *–pm->ndigits* trailing zeros; in other words, rounding occurs *–pm->ndigits* to the left of the decimal point, but the digits rounded away are retained as zeros. The total number of digits required is in *pd->ndigits*. *pd->exponent* always gets 0. Thus if **px* == 12.34 and *pm->ndigits* == –1, then *pd->ds* gets 10, *pd->exponent* gets 0, and *pd->ndigits* gets 2.

pd->more is not used.

econvert(3), fconvert, gconvert, printf(3S), and sprintf, all use double_to_decimal.

SEE ALSO

econvert(3), printf(3S)

NAME

floatingpoint – IEEE floating point definitions

SYNOPSIS

#include <sys/ieeefp.h>
#include <floatingpoint.h>

DESCRIPTION

This file defines constants, types, variables, and functions used to implement standard floating point according to ANSI/IEEE Std 754-1985. The variables and functions are implemented in **libc.a**. The included file **<sys/ieeefp.h>** defines certain types of interest to the kernel.

IEEE Rounding Modes:

fp_direction_type The type of the IEEE rounding direction mode. Note: the order of enumeration varies according to hardware.

fp_direction The IEEE rounding direction mode currently in force. This is a global variable that is intended to reflect the hardware state, so it should only be written indirectly through a function like " .}S 3 2 "**ieee_flags(set,direction,...)**"" "" "" "" "" "" "" that also sets the hardware state.

fp_precision_type The type of the IEEE rounding precision mode, which only applies on systems that support extended precision such as Sun-3 systems with 68881's.

fp_precision The IEEE rounding precision mode currently in force. This is a global variable that is intended to reflect the hardware state on systems with extended precision, so it should only be written indirectly through a function like **ieee_flags("set","precision",...)**.

SIGFPE handling:

sigfpe_code_type The type of a SIGFPE code.

sigfpe_handler_type The type of a user-definable SIGFPE exception handler called to handle a particular SIGFPE code.

SIGFPE_DEFAULT A macro indicating the default SIGFPE exception handling, namely to perform the exception handling specified by calls to **ieee_handler**(3M), if any, and otherwise to dump core using **abort**(3).

SIGFPE_IGNORE A macro indicating an alternate SIGFPE exception handling, namely to ignore and continue execution.

SIGFPE_ABORT A macro indicating an alternate SIGFPE exception handling, namely to abort with a core dump.

IEEE Exception Handling:

N_IEEE_EXCEPTION The number of distinct IEEE floating-point exceptions.

fp_exception_type The type of the N_IEEE_EXCEPTION exceptions. Each exception is given a bit number.

fp_exception_field_type
The type intended to hold at least N_IEEE_EXCEPTION bits corresponding to the IEEE exceptions numbered by **fp_exception_type**. Thus **fp_inexact** corresponds to the least significant bit and **fp_invalid** to the fifth least significant bit. Note: some operations may set more than one exception.

fp_accrued_exceptions
The IEEE exceptions between the time this global variable was last cleared, and the last time a function like **ieee_flags("get","exception",...)** was called to update the variable by obtaining the hardware state.

ieee_handlers An array of user-specifiable signal handlers for use by the standard SIGFPE handler for IEEE arithmetic-related SIGFPE codes. Since IEEE trapping modes correspond to hardware modes, elements of this array should only be modified with a function like **ieee_handler**(3M) that performs the appropriate hardware mode update. If no **sigfpe_handler** has been declared for a particular IEEE-related SIGFPE code, then the related **ieee_handlers** will be invoked.

IEEE Formats and Classification:

single;extended Definitions of IEEE formats.

fp_class_type An enumeration of the various classes of IEEE values and symbols.

IEEE Base Conversion:

The functions described under **floating_to_decimal**(3) and **decimal_to_floating**(3) not only satisfy the IEEE Standard, but also the stricter requirements of correct rounding for all arguments.

DECIMAL_STRING_LENGTH
The length of a **decimal_string**.

decimal_string The digit buffer in a **decimal_record**.

decimal_record The canonical form for representing an unpacked decimal floating-point number.

decimal_form The type used to specify fixed or floating binary to decimal conversion.

decimal_mode A struct that contains specifications for conversion between binary and decimal.

decimal_string_form An enumeration of possible valid character strings representing floating-point numbers, infinities, or NaNs.

FILES

/usr/include/sys/ieeefp.h
/usr/include/floatingpoint.h
/usr/lib/libc.a

SEE ALSO

abort(3), **decimal_to_floating**(3), **econvert**(3), **floating_to_decimal**(3), **ieee_flags**(3M), **ieee_handler**(3M), **sigfpe**(3), **string_to_decimal**(3), **strtod**(3)

NAME

printf, fprintf, sprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>
int printf(format [ , arg ] ... )
char *format;

int fprintf(stream, format [ , arg ] ... )
FILE *stream;
char *format;

int sprintf(s, format [ , arg ] ... )
char *s, *format;

#include <varargs.h>
int _doprnt(format, args, stream)
char *format;
va_list args;
FILE *stream;
```

DESCRIPTION

printf( ) places output on the standard output stream **stdout**. **fprintf**( ) places output on the named output **stream**. **sprintf**( ) places "output", followed by the NULL character (\0), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. **printf**, **fprintf**( ) and **sprintf**( ) return the number of characters transmitted (excluding the NULL character in the case of **sprintf**).

If an output error is encountered **printf**, **fprintf**( ) and **sprintf**( ) return EOF.

Each of these functions converts, formats, and prints its *arg*s under control of the *format*. The *format* is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of zero or more *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '–', described below, has been given) to the field width. The padding is with blanks unless the field width digit string starts with a zero, in which case the padding is with zeros.

A *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string; a NULL digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.

An optional **l** (ell) specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*. An **l** before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision or both may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted. A negative field width argument is taken as a '–' flag followed by a positive field width. If the precision argument is negative, it will be changed to zero.

The flag characters and their meanings are:

– The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (+ or –).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an "alternate form."For **c, d, i, s,** and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x or X** conversion, a non-zero result will have **0x** or **0X** prefixed to it. For **e, E, f, g,** and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,i,o,u,x,X The integer *arg* is converted to signed decimal (**d** or **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a NULL string.

f The float or double *arg* is converted to decimal notation in the style "[–]ddd.ddd" where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e,E The float or double *arg* is converted in the style "[–]d.ddde±ddd," where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** or **E** will be used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

The **e, E, f, g,** and **G** formats print IEEE indeterminate values (infinity or not-a-number) as "Infinity" or "NaN" respectively.

c The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a NULL character (\0) is encountered or until the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first NULL character are printed. A NULL value for *arg* will yield undefined results.

% Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Padding takes place only if the specified field width exceeds the actual width. Characters generated by **printf()** and **fprintf()** are printed as if **putc**(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02," where *weekday* and *month* are pointers to NULL-terminated strings:

```
printf("%s, %s %i, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

NOTE

These routines call **_doprnt**, which is an implementation-dependent routine. Each uses the variable-length argument facilities of **varargs**(3). Although it is possible to use **_doprnt** to take a list of arguments and pass them on to a routine like **printf**, not all implementations have such a routine. We strongly recommend that you use the routines described in **vprintf**(3S) instead.

SEE ALSO

econvert(3), **printf**(3S), **putc**(3S), **scanf**(3V), **varargs**(3), **vprintf**(3S)

BUGS

Very wide fields (>128 characters) fail.

NAME

scanf, fscanf, sscanf – formatted input conversion

SYNOPSIS

#include <stdio.h>

scanf(format [, pointer] ...)
char *format;

fscanf(stream, format [, pointer] ...)
FILE *stream;
char *format;

sscanf(s, format [, pointer] ...)
char *s, *format;

DESCRIPTION

scanf( ) reads from the standard input stream **stdin**. **fscanf( )** reads from the named input stream. **sscanf( )** reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format*, described below, and a set of *pointer* arguments indicating where the converted input should be stored. The results are undefined in there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (SPACE, TAB, NEWLINE, or FORMFEED) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not '**%**'), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character '**%**', an optional assignment suppressing character '*****', an optional numerical maximum field width, an optional **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by '*****'. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c", white space leading an input field is ignored.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion characters are legal:

% A single % is expected in the input at this point; no assignment is done.

d A decimal integer is expected; the corresponding argument should be an integer pointer.

u An unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o An octal integer is expected; the corresponding argument should be an integer pointer.

x A hexadecimal integer is expected; the corresponding argument should be an integer pointer.

i An integer is expected; the corresponding argument should be an integer pointer. It will store the value of the next input item interpreted according to C conventions: a leading "0" implies octal; a leading "0x" implies hexadecimal; otherwise, decimal.

n Stores in an integer argument the total number of characters (including white space) that have been scanned so far since the function call. No input is consumed.

e,f,g A floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is as described for **string_to_decimal**(3), with

fortran_exponent zero.

s A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which will be added automatically. The input field is terminated by a white space character.

c A character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use %1s. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

[Indicates string data; the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last*, thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \0, which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, **x**, and **i** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

Avoid this common error: because **printf**(3V) does not require that the lengths of conversion descriptors and actual parameters match, coders sometimes are careless with the **scanf**() functions. But converting %f to &double or %lf to &float *does not work*; the results are quite incorrect.

scanf() conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

scanf() returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. The constant EOF is returned upon end of input. Note: this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

If the input ends before the first conflict or conversion, EOF is returned. If the input ends after the first conflict or conversion, the number of successfully matched items is returned.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain **thompson\0**. Or:

```
int i, j; float x; char name[50];
(void) scanf("%i%2d%f%*d %[0-9]", &j, &i, &x, name);
```

with input:

011 56789 0123 56a72

will assign **9** to *j*, **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to **getchar( )** (see getc(3S)) will return a. Or:

```
int i, j, s, e; char name[50];
(void) scanf("%i %i %n%s%n", &i, &j, &s, name, &e);
```

with input:

0x11 0xy johnson

will assign **17** to *i*, **0** to *j*, **6** to *s*, will place the string **xy\0** in *name*, and will assign **8** to *e*. Thus, the length of *name* is $e - s = 2$. The next call to **getchar( )** (see getc(3S)) will return a SPACE.

SEE ALSO

getc(3S), printf(3V), stdio(3V), string_to_decimal(3), strtol(3), scanf(3S)

DIAGNOSTICS

These functions return EOF on end of input, and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

CAVEATS

Trailing white space (including a NEWLINE) is left unread unless matched in the control string.

NAME

sigfpe - signal handling for specific SIGFPE codes

SYNOPSIS

#include <signal.h>

#include <floatingpoint.h>

sigfpe_handler_type sigfpe(code, hdl)
sigfpe_code_type code;
sigfpe_handler_type hdl;

DESCRIPTION

This function allows signal handling to be specified for particular SIGFPE codes. A call to sigfpe() defines a new handler *hdl* for a particular SIGFPE *code* and returns the old handler as the value of the function sigfpe. Normally handlers are specified as pointers to functions; the special cases SIGFPE_IGNORE, SIGFPE_ABORT, and SIGFPE_DEFAULT allow ignoring, specifying core dump using abort(3), or default handling respectively.

For these IEEE-related codes:

FPE_FLTINEX_TRAP	fp_inexact - floating inexact result
FPE_FLTDIV_TRAP	fp_division - floating division by zero
FPE_FLTUND_TRAP	fp_underflow - floating underflow
FPE_FLTOVF_TRAP	fp_overflow - floating overflow
FPE_FLTBSUN_TRAP	fp_invalid - branch or set on unordered
FPE_FLTOPERR_TRAP	fp_invalid - floating operand error
FPE_FLTNAN_TRAP	fp_invalid - floating Not-A-Number

default handling is defined to be to call the handler specified to ieee_handler(3M).

For all other SIGFPE codes, default handling is to core dump using abort(3).

The compilation option –ffpa causes fpa recomputation to replace the default abort action for code FPE_FPA_ERROR. Note: SIGFPE_DEFAULT will restore abort rather than FPA recomputation for this code.

Three steps are required to intercept an IEEE-related SIGFPE code with sigfpe:

1) Set up a handler with sigfpe.

2) Enable the relevant IEEE trapping capability in the hardware, perhaps by using assembly-language instructions.

3) Perform a floating-point operation that generates the intended IEEE exception.

Unlike ieee_handler(3M), sigfpe() never changes floating-point hardware mode bits affecting IEEE trapping. No IEEE-related SIGFPE signals will be generated unless those hardware mode bits are enabled.

SIGFPE signals can be handled using sigvec(2), signal(3), sigfpe(3), or ieee_handler(3M). In a particular program, to avoid confusion, use only one of these interfaces to handle SIGFPE signals.

EXAMPLE

A user-specified signal handler might look like this:

```
void sample_handler( sig, code, scp, addr )
        int sig ;           /* sig == SIGFPE always */
        int code ;
        struct sigcontext *scp ;
        char *addr ;
        {
                /*
                  Sample user-written sigfpe code handler.
                  Prints a message and continues.
                  struct sigcontext is defined in <signal.h>.
                 */
                printf(" ieee exception code %x occurred at pc %X \n",code,scp->sc_pc);
        }
```

and it might be set up like this:

```
        extern void sample_handler();
        main()
        {
                sigfpe_handler_type hdl, old_handler1, old_handler2;
        /*
         * save current overflow and invalid handlers; set the new
         * overflow handler to sample_handler() and set the new
         * invalid handler to SIGFPE_ABORT (abort on invalid)
         */
                hdl = (sigfpe_handler_type) sample_handler;
                old_handler1 = sigfpe(FPE_FLTOVF_TRAP, hdl);
                old_handler2 = sigfpe(FPE_FLTOPERR_TRAP, SIGFPE_ABORT);
                ...
        /*
         * restore old overflow and invalid handlers
         */
                sigfpe(FPE_FLTOVF_TRAP,  old_handler1);
                sigfpe(FPE_FLTOPERR_TRAP, old_handler2);
        }
```

FILES

/usr/include/floatingpoint.h
/usr/include/signal.h

SEE ALSO

sigvec(2), abort(3), floatingpoint(3), ieee_handler(3M), signal(3),

DIAGNOSTICS

sigfpe( ) returns BADSIG if *code* is not zero or a defined SIGFPE code.

NAME

signal – simplified software signal facilities

SYNOPSIS

```
#include <signal.h>

void (*signal(sig, func))()
void (*func)();
```

DESCRIPTION

signal() is a simplified interface to the more general **sigvec**(2) facility. Programs that use **signal()** in preference to **sigvec()** are more likely to be portable to all systems.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see **termio**(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the **signal()** call allows signals either to be ignored or to interrupt to a specified location. The following is a list of all signals with names as in the include file **<signal.h>**:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGABRT	6*	abort (generated by **abort**(3) routine)
SIGEMT	7*	emulator trap
SIGFPE	8*	arithmetic exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe or other socket with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop (cannot be blocked)
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
SIGIO	23•	I/O is possible on a descriptor (see **fcntl**(2V))
SIGXCPU	24	cpu time limit exceeded (see **getrlimit**(2))
SIGXFSZ	25	file size limit exceeded (see **getrlimit**(2))
SIGVTALRM	26	virtual time alarm (see **getitimer**(2))
SIGPROF	27	profiling timer alarm (see **getitimer**(2))
SIGWINCH	28•	window changed (see **termio**(4) and **win**(4S))
SIGLOST	29*	resource lost (see **lockd**(8C))
SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

The starred signals in the list above cause a core image if not caught or ignored.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop. If *func* is SIG_IGN the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. **Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.**

If a caught signal occurs during certain system calls, terminating the call prematurely, the call is automatically restarted. In particular this can occur during a **read**(2V) or **write**(2V) on a slow device (such as a terminal; but not a file) and during a **wait**(2).

The value of **signal**() is the previous (or initial) value of *func* for the particular signal.

After a **fork**(2) or **vfork**(2) the child inherits all signals. An **execve**(2) resets all caught signals to the default action; ignored signals remain ignored.

NOTES

The handler routine can be declared:

```
void handler(sig, code, scp, addr)
int sig, code;
struct sigcontext *scp;
char *addr;
```

Here *sig* is the signal number; *code* is a parameter of certain signals that provides additional detail; *scp* is a pointer to the **sigcontext** structure (defined in **<signal.h>**), used to restore the context from before the signal; and *addr* is additional address information. See **sigvec**(2) for more details.

RETURN VALUE

The previous action is returned on a successful call. Otherwise, −1 is returned and **errno** is set to indicate the error.

ERRORS

signal() will fail and no action will take place if one of the following occur:

EINVAL	*sig* is not a valid signal number.
EINVAL	An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.
EINVAL	An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

SEE ALSO

kill(1), **execve**(2), **fork**(2), **getitimer**(2), **getrlimit**(2), **kill**(2V), **ptrace**(2), **read**(2V), **sigblock**(2), **sigpause**(2), **sigsetmask**(2), **sigstack**(2), **sigvec**(2), **vfork**(2), **wait**(2), **write**(2V), **setjmp**(3), **termio**(4)

NAME

string_to_decimal, file_to_decimal, func_to_decimal – parse characters into decimal record

SYNOPSIS

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

void string_to_decimal(pc,nmax,fortran_conventions,pd,pform,pechar)
char **pc;
int nmax;
int fortran_conventions;
decimal_record *pd;
enum decimal_string_form *pform;
char **pechar;

void file_to_decimal(pc,nmax,fortran_conventions,pd,pform,pechar,pf,pnread)
char **pc;
int nmax;
int fortran_conventions;
decimal_record *pd;
enum decimal_string_form *pform;
char **pechar;
FILE *pf;
int *pnread;

void func_to_decimal(pc,nmax,fortran_conventions,pd,pform,pechar,pget,pnread,punget)
char **pc;
int nmax;
int fortran_conventions;
decimal_record *pd;
enum decimal_string_form *pform;
char **pechar;
int (*pget)();
int *pnread;
int (*punget)();
```

DESCRIPTION

The **char_to_decimal** functions parse a numeric token from at most *nmax* characters in a string ***pc* or file **pf* or function (*pget)() into a decimal record **pd*, classifying the form of the string in **pform* and **pechar*. The accepted syntax is intended to be sufficiently flexible to accomodate many languages:

whitespace value

or

whitespace sign value

where *whitespace* is any number of characters defined by *isspace* in **/usr/include/ctype.h**, *sign* is either of [+–], and *value* can be *number*, *nan*, or *inf*. *inf* can be INF (*inf_form*) or INFINITY (*infinity_form*) without regard to case. *nan* can be NAN (*nan_form*) or NAN(nstring) (*nanstring_form*) without regard to case; nstring is any string of characters not containing ')' or NULL; nstring is copied to pd->ds and, currently, not used subsequently. *number* consists of

significand

or

significand efield

where *significand* must contain one or more digits and may contain one point; possible forms are

digits	*(int_form)*
digits.	*(intdot_form)*
.digits	*(dotfrac_form)*
digits.digits	*(intdotfrac_form)*

efield consists of

echar digits

or

echar sign digits

where *echar* is one of [Ee], and *digits* contains one or more digits.

When *fortran_conventions* is nonzero, additional input forms are accepted according to various Fortran conventions:

0 no Fortran conventions
1 Fortran list-directed input conventions
2 Fortran formatted input conventions, ignore blanks (BN)
3 Fortran formatted input conventions, blanks are zeros (BZ)

When *fortran_conventions* is nonzero, *echar* may also be one of [Dd], and *efield* may also have the form

sign digits

When *fortran_conventions*>= 2, blanks may appear in the *digits* strings for the integer, fraction, and exponent fields and may appear between *echar* and the exponent sign and after the infinity and NaN forms. If *fortran_conventions*== 2, the blanks are ignored. When *fortran_conventions*== 3, the blanks that appear in *digits* strings are interpreted as zeros, and other blanks are ignored.

The form of the accepted decimal string is placed in **peform*. If an *efield* is recognized, **pechar* is set to point to the *echar*.

On input, **pc* points to the beginning of a character string buffer of length >= *nmax*. On output, **pc* points to a character in that buffer, one past the last accepted character. **string_to_decimal()** gets its characters from the buffer; **file_to_decimal()** gets its characters from **pf* and records them in the buffer, and places a null after the last character read. **func_to_decimal()** gets its characters from an int function *(*pget)()*.

The scan continues until no more characters could possibly fit the acceptable syntax or until *nmax* characters have been scanned. If the *nmax* limit is not reached then at least one extra character will usually be scanned that is not part of the accepted syntax. **file_to_decimal()** and **func_to_decimal()** set **pnread* to the number of characters read from the file; if greater than *nmax*, some characters were lost. If no characters were lost, **file_to_decimal()** and **func_to_decimal()** attempt to push back, with **ungetc**(3S) or **(*punget)()**, as many as possible of the excess characters read, adjusting **pnread* accordingly. If all unget calls are successful, then ***pc* will be NULL. No push back will be attempted if **(*punget)()** is NULL.

Typical declarations for **pget()** and **punget()** are:

```
int xget()
{ ... }
int (*pget)() = xget ;
int xunget(c)
char c ;
{ ... }
int (*punget)() = xunget ;
```

If no valid number was detected, *pd->fpclass* is set to **fp_signaling**, **pc* is unchanged, and **pform* is set to **invalid_form**.

NAME

IEEE environment - mode, status, and signal handling subprograms for IEEE arithmetic

SYNOPSIS

#include <f77/f77_floatingpoint.h>

integer function ieee_flags(action,mode,in,out)
character*(*) action, mode, in, out

integer function ieee_handler(action,exception,hdl)
character*(*) action, exception
sigfpe_handler_type hdl

sigfpe_handler_type function sigfpe(code, hdl)
sigfpe_code_type code
sigfpe_handler_type hdl

DESCRIPTION

These subprograms provide modes and status required to fully exploit ANSI/IEEE Std 754-1985 arithmetic in a Fortran program. They correspond closely to the functions *ieee_flags(3M)*, *ieee_handler(3M)*, and *sigfpe(3)*.

EXAMPLES

The following examples illustrate syntax.

```
integer ieeer
character*1 mode, out, in
ieeer = ieee_flags('clearall',mode, in, out)
```

sets ieeer to 0, rounding direction to 'nearest', rounding precision to 'extended', and all accrued exception-occurred status to zero.

```
character*1 out, in
ieeer = ieee_flags('clear','direction', in, out)
```

sets ieeer to 0, and rounding direction to 'nearest'.

```
character*1 out
ieeer = ieee_flags('set','direction','tozero',out)
```

sets ieeer to 0 and the rounding direction to 'tozero' unless the hardware does not support directed rounding modes; then ieeer is set to 1.

```
character*16 out
ieeer = ieee_flags('clear','exception','all',out)
```

sets ieeer to 0 and clears all accrued exception-occurred bits. If subsequently overflow, invalid, and inexact exceptions are generated then

```
character*16 out
ieeer = ieee_flags('get','exception','overflow',out)
```

sets ieeer to 25 and out to 'overflow'.

A user-specified signal handler might look like this:

```
      integer function sample_handler ( sig, code, sigcontext )
      integer sig
      integer code
      integer sigcontext(5)
c     Sample user-written sigfpe code handler.
c     Prints a message and terminates.
c     sig .eq. SIGFPE always.
c     The structure of sigcontext is defined in <signal.h>.
      print *,' ieee exception code ',code,' occurred at pc ',sigcontext(4)
      call abort(' ieee exception occurred ')
      end
```

and it might be set up like this:

```
      extern sample_handler
      integer ieeer
      ieeer = ieee_handler ( 'set', 'overflow', sample_handler )
      if (ieeer .ne. 0) print *,' ieee_handler can not set overflow '
```

FILES

/usr/include/f77/f77_floatingpoint.h
/usr/lib/libm.a

SEE ALSO

floatingpoint(3), signal(3), sigfpe(3), f77_floatingpoint(3F), ieee_flags(3M), ieee_handler(3M)