This chapter describes how you can perform the conversions required by the IEEE standard using MathLib C functions. For each type of conversion, this chapter lists the functions you can use to perform that conversion. It shows the declarations of these functions, describes what they do, describes when they raise floating-point exceptions, and gives examples of how to use them. For a description of the conversions required by the IEEE standard and the details of how each conversion is performed in PowerPC Numerics, see Chapter 5, "Conversions." All of the conversion function declarations appear in the file `fp.h`.

Converting Floating-Point to Integer Formats

In C, the default method of converting floating-point numbers to integers is to simply discard the fractional part (truncate). MathLib provides two functions that convert floating-point numbers to integers using methods other than the default C method and that return the integers in integer types.

rinttol(x)	Returns the nearest integer to x in the current rounding direction as an integer type.
roundtol(x)	Adds 1/2 to the magnitude of x, chops to an integer, and returns the value as an integer type.

rinttol

You can use the `rinttol` function to round a real number to the nearest integer in the current rounding direction.

```
long int rinttol (double_t x);
```

`x`	Any floating-point number.

DESCRIPTION

The `rinttol` function rounds its argument to the nearest integer in the current rounding direction and places the result in a `long int` type. The available rounding directions are upward, downward, to nearest, and toward zero.

The `rinttol` function provides the floating-point to integer conversion as described in the IEEE standard. It differs from `rint` (described on page 6-13) in that it returns the value in an integer type; `rint` returns the value in a floating-point type.

EXCEPTIONS

When x is finite and nonzero, either the result of $\text{rinttol}(x)$ is exact or it raises one of the following exceptions:

- inexact (if x is not an integer)
- invalid (if the integer result is outside the range of the `long int` type)

SPECIAL CASES

Table 9-1 shows the results when the argument to the `rinttol` function is a zero, a NaN, or an Infinity.

Table 9-1 Special cases for the `rinttol` function

Operation	Result	Exceptions raised
rinttol(+0)	+0	None
rinttol(–0)	–0	None
rinttol(NaN)	Undefined	None*
rinttol(+∞)	Undefined	Invalid
rinttol(–∞)	Undefined	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = rinttol(+INFINITY);/* z = unspecified value for all rounding
                         directions because +INFINITY exceeds the
                         range of long int. The invalid exception
                         is raised. */
z = rinttol(300.1);    /* z = 301 if rounding direction is upward
                         else z = 300. The inexact exception is
                         raised.*/
z = rinttol(-300.1);   /* z = -301 if rounding direction is
                         downward else z = -300. The inexact
                         exception is raised. */
```

roundtol

You can use the `roundtol` function to round a real number to the nearest integer value by adding 1/2 to the magnitude and truncating.

```
long int roundtol (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `roundtol` function adds 1/2 to the magnitude of its argument and chops to integer, returning the answer in `long int` type.

The result is returned in an integer data type. (The return type is the difference between the `roundtol` function and the `round` function, described on page 9-10.)

This function is not affected by the current rounding direction. Notice that the `roundtol` function rounds halfway cases (1.5, 2.5, and so on) away from 0. With the default rounding direction, `rinttol` (described on page 9-3) rounds halfway cases to the even integer.

EXCEPTIONS

When x is finite and nonzero, either the result of $\text{roundtol}(x)$ is exact or it raises one of the following exceptions:

- inexact (if x is not an integer)
- invalid (if the integer result is outside the range of the `long int` type)

SPECIAL CASES

Table 9-2 shows the results when the argument to the `roundtol` function is a zero, a NaN, or an Infinity.

Table 9-2 Special cases for the `roundtol` function

Operation	Result	Exceptions raised
$\text{roundtol}(+0)$	+0	None
$\text{roundtol}(-0)$	–0	None
$\text{roundtol}(\text{NaN})$	Undefined	None*
$\text{roundtol}(+\infty)$	Undefined	Invalid
$\text{roundtol}(-\infty)$	Undefined	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = roundtol(+INFINITY);   /* z = an unspecified value because
                              +∞ is outside of the range of long
                              int. */
z = roundtol(0.5);         /* z = 1 because |0.5| + 0.5 = 1.0. The
                              inexact exception is raised. */
z = roundtol(–0.9);        /* z = –1 because |–0.9| + 0.5 = 1.4.
                              The inexact exception is raised. */
```

Rounding Floating-Point Numbers to Integers

MathLib provides six functions that convert floating-point numbers to integers and return the integer in a floating-point type. The first is the `rint` function, which performs the round-to-integer operation as described in Chapter 6, "Numeric Operations and Functions." The other functions either round in a specific direction or perform a variation of the `rint` operation.

ceil(x)	Returns the nearest integer not less than x.
floor(x)	Returns the nearest integer not greater than x.
nearbyint(x)	Returns the nearest integer to x in the current rounding direction.
round(x)	Adds 1/2 to the magnitude of x and chops to an integer.
trunc(x)	Truncates the fractional part of x.

ceil

You can use the `ceil` function to round a real number upward to the nearest integer value.

```
double_t ceil (double_t x);
```

`x`	Any floating-point number.

DESCRIPTION

The `ceil` function rounds its argument upward. This is an ANSI standard C library function. The result is returned in a floating-point data type.

This function is the same as performing the following code sequence:

```
r = fegetround();       /* save current rounding direction */
fesetround(FE_UPWARD);  /* round upward */
rint(x);                /* round to integer */
fesetround(r);          /* restore rounding direction */
```

EXCEPTIONS

When x is finite and nonzero, the result of $\text{ceil}(x)$ is exact.

SPECIAL CASES

Table 9-3 shows the results when the argument to the `ceil` function is a zero, a NaN, or an Infinity.

Table 9-3 Special cases for the `ceil` function

Operation	Result	Exceptions raised
$\text{ceil}(+0)$	+0	None
$\text{ceil}(-0)$	–0	None
$\text{ceil}(\text{NaN})$	NaN	None*
$\text{ceil}(+\infty)$	$+\infty$	None
$\text{ceil}(-\infty)$	$-\infty$	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = ceil(+INFINITY); /* z = +INFINITY because +INFINITY is already
                        an integer value by definition. */
z = ceil(300.1);     /* z = 301.0 */
z = ceil(-300.1);    /* z = -300.0 */
```

floor

You can use the `floor` function to round a real number downward to the next integer value.

```
double_t floor (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `floor` function rounds its argument downward. This is an ANSI standard C library function. The result is returned in a floating-point data type.

This function is the same as performing the following code sequence:

```
r = fegetround();          /* save current rounding direction */
fesetround(FE_DOWNWARD);   /* round downward */
rint(x);                   /* round to integer */
fesetround(r);             /* restore rounding direction */
```

EXCEPTIONS

When x is finite and nonzero, the result of $\mathrm{floor}(x)$ is exact.

SPECIAL CASES

Table 9-4 shows the results when the argument to the `floor` function is a zero, a NaN, or an Infinity.

Table 9-4 Special cases for the `floor` function

Operation	Result	Exceptions raised
floor(+0)	+0	None
floor(–0)	–0	None
floor(NaN)	NaN	None*
floor(+∞)	+∞	None
floor(–∞)	–∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = floor(+INFINITY);   /* z = +INFINITY because +∞ is already an
                           integer value by definition. */
z = floor(300.1);       /* z = 300.0 */
z = floor(-300.1);      /* z = -301.0 */
```

nearbyint

You can use the `nearbyint` function to round a real number to the nearest integer in the current rounding direction.

```
double_t nearbyint (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `nearbyint` function rounds its argument to the nearest integer in the current rounding direction. The available rounding directions are upward, downward, to nearest, and toward zero.

The `nearbyint` function provides the floating-point to integer conversion described in the IEEE Standard 854. It differs from `rint` (described on page 6-13) only in that it does not raise the inexact flag when the argument is not already an integer.

EXCEPTIONS

When x is finite and nonzero, the result of $\text{nearbyint}(x)$ is exact.

SPECIAL CASES

Table 9-5 shows the results when the argument to the `nearbyint` function is a zero, a NaN, or an Infinity.

Table 9-5 Special cases for the `nearbyint` function

Operation	Result	Exceptions raised
$\text{nearbyint}(+0)$	+0	None
$\text{nearbyint}(-0)$	–0	None
$\text{nearbyint}(\text{NaN})$	NaN	None*
$\text{nearbyint}(+\infty)$	+∞	None
$\text{nearbyint}(-\infty)$	–∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

Conversion Functions

EXAMPLES

```
z = nearbyint(+INFINITY);  /* z = +INFINITY for all rounding
                              directions. */
z = nearbyint(300.1);      /* z = 301.0 if rounding direction is
                              upward, else z = 300.0. */
z = nearbyint(–300.1);     /* z = –301.0 if rounding direction is
                              downward, else z = –300.0. */
```

round

You can use the `round` function to round a real number to the integer value obtained by adding 1/2 to the magnitude and truncating.

```
double_t round (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `round` function adds 1/2 to the magnitude of its argument and chops to integer. The result is returned in a floating-point data type.

This function is not affected by the current rounding direction. Notice that the `round` function rounds halfway cases (1.5, 2.5, and so on) away from 0. With the default rounding direction, `rint` (described on page 6-13) rounds halfway cases to the even integer.

EXCEPTIONS

When x is finite and nonzero, either the result of $\mathrm{round}(x)$ is exact or it raises the following exception:

- inexact (if x is not an integer value)

SPECIAL CASES

Table 9-6 shows the results when the argument to the `round` function is a zero, a NaN, or an Infinity.

Table 9-6 Special cases for the `round` function

Operation	Result	Exceptions raised
round(+0)	+0	None
round(–0)	–0	None
round(NaN)	NaN	None*
round(+∞)	+∞	None
round(–∞)	–∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = round(+INFINITY);   /* z = +INFINITY because +∞ is already an
                           integer value by definition. */
z = round(0.5);         /* z = 1.0 because |0.5| + 0.5 = 1.0. The
                           inexact exception is raised. */
z = round(-0.9);        /* z = –1.0 because |–0.9| + 0.5 = 1.4.
                           The inexact exception is raised. */
```

trunc

You can use the `trunc` function to truncate the fractional part of a real number so that just the integer part remains.

```
double_t trunc (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `trunc` function chops off the fractional part of its argument. This is an ANSI standard C library function.

This function is the same as performing the following code sequence:

```
r = fegetround();          /* save current rounding direction */
fesetround(FE_TOWARDZERO); /* round toward zero */
rint(x);                   /* round to integer */
fesetround(r);             /* restore rounding direction */
```

EXCEPTIONS

When x is finite and nonzero, the result of $\text{trunc}(x)$ is exact.

SPECIAL CASES

Table 9-7 shows the results when the argument to the `trunc` function is a zero, a NaN, or an Infinity.

Table 9-7 Special cases for the `trunc` function

Operation	Result	Exceptions raised
$\text{trunc}(+0)$	+0	None
$\text{trunc}(-0)$	–0	None
$\text{trunc}(\text{NaN})$	NaN	None*
$\text{trunc}(+\infty)$	+∞	None
$\text{trunc}(-\infty)$	–∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = trunc(+INFINITY);   /* z = +INFINITY because +∞ is already an
                           integer value by definition. */
z = trunc(300.1);       /* z = 300.0 */
z = trunc(–300.1);      /* z = –300.0 */
```

Converting Integers to Floating-Point Formats

In the C programming language, conversions from integers stored in an integer format to floating-point formats are automatic when you assign an integer to a floating-point variable.

```
double d;
int x = 1;
d = x;      /* value 1 automatically converted to double format */
```

Converting Between Floating-Point Formats

In the C programming language, conversions between floating-point formats are automatic when you assign a floating-point number of one type to a variable of another type.

```
float f = 0.0f;         /* single format */
double d = 1.1;
long double ld;         /* double-double format */

f = d;      /* double 1.1 converted to single format */
ld = f;     /* single 1.1 converted to double-double format */
d = ld;     /* double-double 1.1 converted to double format */
```


Converting Between Binary and Decimal Numbers

MathLib provides two functions that let you manually convert between binary and decimal formats.

`dec2num` Converts a decimal number to a binary number.

`num2dec` Converts a binary number to a decimal number.

Conversions between binary floating-point numbers and decimal numbers use structures of type `decimal`. The `decimal` structure is defined in the header file `fp.h` as

```
struct decimal
{
   char sgn;
   char unused;
   short exp;
   struct
   {
      unsigned char length;
      unsigned char text[SIGDIGLEN];
      unsigned char unused;
   } sig;
} decimal;
```

`sgn` The sign of the number (0 is positive, 1 is negative).

`exp` The exponent of the number. The exponent is expressed as a power of 10.

`sig` The significand. String `sig.text` contains the significand as a decimal integer in the form of a string, that is, with the string length in the zeroth byte (`sig.length`) and the initial character of the string in the first byte (`sig.text[0]` to `sig.text[SIGDIGLEN – 1]`).

The value represented is

$$(-1)^{\mathtt{sgn}} \times \mathtt{sig} \times 10^{\mathtt{exp}}$$

For example, if `sgn` equals 1, `exp` equals –3, and `sig` equals “85” (string length `sig.length` equals 2, not shown), then the number represented is –0.085.

Note

The maximum length of the string `sig` is implementation dependent. The limit is 36 characters. Also, the representations of 0 and 1 in the 16-bit word `sgn` are implementation dependent. ◆

Conversions from binary to decimal use a decimal format structure to specify how the number should look in decimal. The `decform` structure is defined in the header file `fp.h` as

```
struct decform
{
   char style; /* FLOATDECIMAL or FIXEDDECIMAL */
   char unused;
   short digits;
} decform;
```

`style` The style of output. This field equals 0 (`FLOATDECIMAL`) for floating and 1 (`FIXEDDECIMAL`) for fixed.

`digits` The number of significant digits for the floating style and the number of digits to the right of the decimal point for the fixed style. (The value of `digits` may be negative if the style is fixed.)

Note

Formatting details, such as the representations of 0 and 1 in the 16-bit `style` word, are implementation dependent. ◆

If the `style` field of the `decform` structure equals 0 (in C, `f.style == FLOATDECIMAL`), the output is formatted in floating style, with the `digits` field specifying the number of significant digits required. Output in floating style is represented in the following format; Table 9-8 defines its components.

`[– |     ]`*m*`[.`*nnn*`]e[+ | -]`*dddd*

Table 9-8 Format of decimal output string in floating style

Component	Description
Minus sign (–) or space	Minus sign if `sgn` = 1; space if `sgn` = 0
m	Single digit, 0 only if value represented is 0
Point (.)	Present if `digits` > 1
nnn	String of digits; present if `digits` > 1
`e`	The letter *e*
Plus sign (+) or minus sign (–)	Plus sign if `exp` ≥ 0; minus sign if `exp` < 0.
dddd	One to four exponent digits

If the `style` field of the `decform` structure equals 1 (in C, `f.style == FIXEDDECIMAL`), the output is formatted in fixed style, with the `digits` field specifying the number of digits to follow the decimal point. All output in fixed style is represented in the following format; Table 9-9 defines its components.

`[-]`*mmm*`[.`*nnn*`]`

Table 9-9 Format of decimal output string in fixed style

Component	Description
Minus sign (–)	Present if `sgn` = 1
mmm	String of digits; at least one digit but no superfluous leading zeros
Point (.)	Present if `digits` > 0
nnn	String of digits of length equal to `digits`; present if `digits` > 0

Note that if `sgn` equals 0, then floating-style output begins with a space but fixed-style output does not.

Double-double values being converted to decimal strings are first rounded to 113 bits (if they in fact span more than that number of bits in their significands) and then converted to the decimal string of the desired length.

dec2num

You can use the `dec2num` function to convert a decimal number to a binary floating-point number.

```
float dec2f (const decimal *d);
double_t dec2num (const decimal *d);
long double dec2numl (const decimal *d);
short int dec2s (const decimal *d);
long int dec2l (const decimal *d);
```

`d` The `decimal` structure to be converted. See page 9-13 for the definition of the `decimal` structure.

DESCRIPTION

The `dec2num` function converts a decimal number in a `decimal` structure to a double format floating-point number. Conversions from the `decimal` structure type handle any `sig` string of length 36 or less (with an implicit decimal point at the right end).

There are three versions of this function that convert to a floating-point type: `dec2f` converts the decimal number to the `float` type, `dec2num` converts to the `double` type, and `dec2numl` converts to the `long double` type. The other two versions of this function, `dec2s` and `dec2l`, convert to the short and long integer types, respectively.

IMPORTANT

When you create a `decimal` structure, you must set `sig.length` to the size of the string you place in `sig.text`. You cannot leave the `length` field undefined. ▲

Before using this function, you can use the numeric formatter (`str2dec`, described on page 9-21) to convert a decimal string to a decimal structure suitable for input to the `dec2num` function.

EXCEPTIONS

When the `sig` string is longer than 36 characters, the result is undefined.

SPECIAL CASES

The following special cases apply:

- If `sig.text[0]` is “0” (zero), the `decimal` structure is converted to zero. For example, a `decimal` structure with `sig` = “0913” is converted to zero.

- If `sig.text[0]` is "N", the `decimal` structure is converted to a NaN. The succeeding characters of `sig` are interpreted as a hexadecimal representation of the result's significand: if fewer than four characters follow the N, then they are right aligned in the high-order 15 bits of the field *f* illustrated in the section "Formats" in Chapter 2, "Floating-Point Data Formats"; if four or more characters follow the N, then they are left aligned in the result's significand.
- If `sig.text[0]` is "I", the `decimal` structure is converted to an Infinity.

EXAMPLES

```
decimal d;
double_t result;

d.sgn = 0;
d.exp = 3;
d.sig.length = 3;
d.sig.text[0] = '2';
d.sig.text[1] = '0';
d.sig.text[2] = '8';
result = dec2num(&d);      /* result = 208,000 stored in double
                              format */
```


num2dec

You can use the `num2dec` function to convert a binary floating-point number to a decimal number.

```
void num2dec (const decform *f, double_t x, decimal *d);
void num2decl (const decform *f, long double x, decimal *d);
```

`f`	A `decform` structure that describes how the number should look in decimal. See page 9-14 for a description of the `decform` structure.
`x`	The floating-point number to be converted.
`d`	Upon return, a pointer to the `decimal` structure containing the number. See page 9-13 for a description of the `decimal` structure.

DESCRIPTION

The `num2dec` function converts a floating-point number to a decimal number. The decimal number is contained in a `decimal` structure. Each conversion to a `decimal` structure `d` is controlled by a `decform` structure `f`. All implementations allow 36 digits to be returned in the `sig` field of the `decimal` structure. The implied decimal point is at the right end of `sig`, with `exp` set accordingly.

After using the `num2dec` function, you can use the `dec2str` function to convert the `decimal` structure to a character string.

IMPORTANT

Use the same decimal format structure settings for `dec2str` as you used for `num2dec`; otherwise, the results are unspecified. ▲

EXCEPTIONS

When the number of digits specified in a `decform` structure exceeds an implementation maximum (which is 36), the result is undefined.

A number might be too large to represent in a chosen fixed style. For instance, if the implementation's maximum length for `sig` is 36, then 10^{35} (which requires 33 digits to the left of the point in fixed-style representations) is too large for a fixed-style representation specifying more than two digits to the right of the point. If a number is too large for a chosen fixed style, then (depending on the numeric implementation) one of two results is returned: an implementation might return the most significant digits of the number in `sig` and set `exp` so that the `decimal` structure contains a valid floating-style approximation of the number; alternatively, an implementation might simply set `sig` to the string "?". Note that in any implementation, the following test determines whether a nonzero finite number is too large for the chosen fixed style.

```
decimal d;
decform f;
int too_big; /* Boolean */

too_big = (-d.exp != f.digits) || (d.sig.text[0] == "?");
```

For fixed-point formatting, PowerPC Numerics treats a negative value for `digits` as a specification for rounding to the left of the decimal point; for example, `digits` = –2 means to round to hundreds. For floating-point formatting, a negative value for `digits` gives unspecified results.

SPECIAL CASES

- For zeros, the character "0" is placed in `sig.text[0]`.
- For NaNs, The character "N" is placed in `sig.text[0]`. The character "N" might be followed by a hexadecimal representation of the input significand. The third and fourth hexadecimal digits following the "N" give the NaN code. For example, "N4021000000000000" has NaN code 0x21.
- For Infinities, the character "I" is placed in `sig.text[0]`.

In all three of these cases, `exp` is undefined.

EXAMPLES

```
decimal d;
decform f;
double_t fp_num = 1.000007;

f.style = FLOATDECIMAL;     /* floating-point format */
f.digits = 7;               /* seven significant digits */
num2dec(&f, fp_num, &d);    /* d now contains 1.000007 expressed
                               in decimal structure */
```

Converting Between Decimal Formats

MathLib provides a scanner for converting from decimal strings to `decimal` structures and a formatter for converting from `decimal` structures to decimal strings.

`dec2str` Converts `decimal` structures to decimal strings. The PowerPC Numerics formatter.

`str2dec` Converts decimal strings to `decimal` structures. The PowerPC Numerics scanner.

dec2str

You can use the `dec2str` function to convert a number in a `decimal` structure to a decimal string.

```
void dec2str (const decform *f, const decimal *d, char *s);
```

`f` A `decform` structure that describes how the number should look in decimal. See page 9-14 for a description of the `decform` structure.

`d` The `decimal` structure to be converted. See page 9-13 for the definition of the `decimal` structure.

`s` On return, a string representing the number in decimal.

DESCRIPTION

The `dec2str` function is the PowerPC Numerics formatter. It takes a number from a decimal structure and converts it to a string. You can use the `num2dec` function to convert a binary floating-point number to a `decimal` structure appropriate for input to the `dec2str` function.

IMPORTANT

Use the same decimal format structure settings for `dec2str` as you used for `num2dec`; otherwise, results are unspecified. ▲

The numeric formatter is controlled by a `decform` structure `f`. With floating style, numbers formatted using the same value for `f.digits` have aligning decimal points and *e*'s. To ensure that numbers have the same width also, pad the exponent-digits field with spaces to a width of 4. For example, if `f.digits` = 12, then pad 12 + 8 – length(`s`) spaces on the right of the result string `s`. The value 8 accounts for the sign, point, letter *e*, exponent sign, and four exponent digits. Note that this scheme gives the correct field width for NaNs and Infinities too.

With fixed style, numbers formatted using the same value for `f.digits` have aligning decimal points if enough leading spaces are added to the result string `s` to attain a fixed width, which must be no narrower than the widest `s`.

IMPORTANT

When you create a `decimal` structure, you must set `sig.length` to the size of the string you place in `sig.text`. You cannot leave the `length` field undefined. ▲

EXCEPTIONS

The formatter is always exact and signals no exceptions.

SPECIAL CASES

For fixed-point formatting, `dec2str` treats a negative value for `digits` as a specification for rounding to the left of the decimal point; for example, `digits` = –2 means to round to hundreds. For floating-point formatting, values for `digits` less than 1 are treated as 1.

NaNs are formatted as `NAN`; Infinities are formatted as `INF`. A leading sign or space is included according to the style convention.

The formatter never returns fewer significant digits than are contained in `sig`. However, if the `decform` structure calls for more significant digits than are contained in `sig`, then the formatter pads with zeros as needed.

If more than 80 characters are required to honor `digits`, then the formatter returns the string "?".

EXAMPLES

Suppose you have an accounting program that computes exact values using binary numbers of pennies and prints outputs in dollars and cents. If you simply divide the number of pennies by 100 to get dollars, you incur errors because hundredths are not exact in binary. One way to print out exact values in dollars and cents is to convert the number of pennies to a `decimal` structure, perform the division by adjusting the exponent, and print the result, as shown in Listing 9-1.

Listing 9-1 Accounting program

```
#include <fp.h>

   decform     df;
   double      pennies;    /* This is the input value */
   decimal     dpennies;   /* decimal value for pennies */
   char *      dollars;    /* string to print as $$$.¢¢ */

{
   df.style = FIXEDDECIMAL;
   df.digits = 0;    /* start with 0 digits after decimal point */

   num2dec(&df, pennies, &dpennies);         /* decimal pennies */
   dpennies.exp = dpennies.exp – 2;          /* divide by 100 */

   df.digits = 2;      /* request 2 digits after decimal point */
   dec2str(&df, &dpennies, dollars);
   /* dollar string to print */
}
```


str2dec

You can use the `str2dec` function to convert a decimal string to a `decimal` structure.

```
void str2dec (const char *s, short *ix, decimal *d, short *vp);
```

`s`	The character string containing the number to be converted.
`ix`	On entry, the starting position in the string. On return, one greater than the position of the last character in the string that was parsed if the entire string was not converted successfully.
`d`	On return, a pointer to the `decimal` structure containing the decimal number. See page 9-13 for a description of the `decimal` structure.
`vp`	On return, a Boolean argument indicating the success of the function. If the entire string was parsed, `vp` is true. If part of the string was parsed, `vp` is false and `ix` indicates where the function stopped parsing.

DESCRIPTION

The `str2dec` function is the PowerPC Numerics scanner, which is designed for use both with fixed strings and with strings being received interactively character by character. The scanner parses the longest possible numeric substring; if no numeric substring is recognized, then the value of `ix` remains unchanged.

To convert floating-point strings embedded in text, parse to the beginning of a floating-point string ([+ | –] *digit*) and pass the current scan location as the index into the text. The conversion routine will return the value scanned and a new value of the index for continued parsing.

You might need to distinguish those numeric ASCII strings that represent values of an integer format. You can do this by scanning the source, looking for integer syntax. You can handle integers yourself and send to the numeric scanner any strings with floating-point syntax (that is, containing a period (.), an E, or an e). You might also want to pass along to the scanner any strings that cause integer overflow.

EXCEPTIONS

The scanner signals no exceptions. It faithfully converts all values within range that are representable in the `decimal` structure format.

SPECIAL CASES

To convert a zero, NaN, or Infinity, use one of the following as input:

```
–0    +0    0    –INF    Inf    NAN    –NaN()    nan
```

EXAMPLES

Listing 9-2 shows an example of how to scan decimal strings into an application and then convert the strings to binary floating-point numbers using MathLib functions. Table 9-10 shows some sample inputs to the loop shown in Listing 9-2 and the results after each string has been converted to a decimal structure using `str2dec`.

Listing 9-2 Scanning algorithm

```
s = "";        /* initialize string */

/* loop until string is not a valid prefix*/
do
  {

   /* code to get next character and append to string goes here */

   /* scan string */
  ix = 0;
  str2dec(s, &ix, &d, &vp);
  }
while (vp = false);

/* convert from decimal to numeric-format result */
result = dec2num(d);
```

Table 9-10 Examples of conversions to `decimal` structures

Input string	Index In	Index Out	Output value	Valid prefix
12	0	2	12	True
12E	0	2	12	True
12E–	0	2	12	True
12E–3	0	5	12E–3	True
12E–X	0	2	12	False
12E–3X	0	5	12E–3	False
x12E–3	1	6	12E–3	True
IN	0	0	NAN	True
INF	0	3	INF	True

Conversions Summary

This section summarizes the C constants, macros, functions, and type definitions associated with converting floating-point values.

C Summary

Constants

```
#define      SIGDIGLEN      36            /* significant decimal digits */
#define      DECSTROUTLEN   80            /* max length for dec2str output */
#define      FLOATDECIMAL   ((char)(0))
#define      FIXEDDECIMAL   ((char)(1))
```

Data Types

```
struct decimal
{
   char sgn;                              /* sign 0 for +, 1 for – */
   char unused;
   short exp;                             /* decimal exponent */
   struct
   {
      unsigned char length;
      unsigned char text[SIGDIGLEN];      /* significant digits */
      unsigned char unused;
   } sig;
};
typedef struct decimal decimal;

struct decform
{
   char style;                         /* FLOATDECIMAL or FIXEDDECIMAL */
   char unused;
   short digits;
};
typedef struct decform decform;
```

Conversion Routines

Converting Floating-Point Formats to Integer Formats

```
long int rinttol             (double_t x);
long int roundtol            (double_t x);
```

Rounding Floating-Point Numbers to Integers

```
double_t ceil                (double_t x);
double_t floor               (double_t x);
double_t nearbyint           (double_t x);
double_t round               (double_t x);
double_t trunc               (double_t x);
```

Converting Decimal Numbers to Binary Numbers

```
float dec2f                  (const decimal *d);
double_t dec2num             (const decimal *d);
long double dec2numl         (const decimal *d);
short int dec2s              (const decimal *d);
long int dec2l               (const decimal *d);
```

Converting Binary Numbers to Decimal Numbers

```
void num2dec                 (const decform *f, double_t x, decimal *d);
void num2decl                (const decform *f, long double x, decimal *d);
```

Converting Between Decimal Formats

```
void dec2str                 (const decform *f, const decimal *d, char *s);
void str2dec                 (const char *s, short *ix, decimal *d,
                              short *vp);
```