This chapter describes how to control the floating-point environment using functions defined in MathLib.

As described in Chapter 4, “Environmental Controls,” the rounding direction and the exception flags are the parts of the environment that you can access. You can test and change the rounding direction, and you can test, set, and clear the exceptions flags. You may also save and restore both the rounding direction and exception flags together as a single entity. This chapter describes the functions that perform these tasks. For the definitions of rounding direction and exception flags, see Chapter 4.

Read this chapter to learn how to access and manipulate the floating-point environment in the C language. All of the environmental control function declarations appear in the file `fenv.h`.

IMPORTANT

If your compiler supports the environmental access switch described in Appendix D, “FPCE Recommendations for Compilers,” the switch must be turned on in the program before you use any of the functions described in this chapter. ▲

Controlling the Rounding Direction

In MathLib, the following functions control the rounding direction:

`fegetround`	Returns the current rounding direction.
`fesetround`	Sets the rounding direction.

The four rounding direction modes are defined as the constants shown in Table 8-1.

Table 8-1 Rounding direction modes in MathLib

Rounding direction	Constant
To nearest	`FE_TONEAREST`
Toward zero	`FE_TOWARDZERO`
Upward	`FE_UPWARD`
Downward	`FE_DOWNWARD`

fegetround

You can use the `fegetround` function to save the current rounding direction.

```
int fegetround (void);
```

DESCRIPTION

The fegetround function returns an integer that specifies which rounding direction is currently being used. The integer it returns will be equal to one of the constants shown in Table 8-1. You can save the returned value in an integer variable to save the current rounding direction.

EXAMPLES

```
int rounddir;
double_t x, y, result;

rounddir = fegetround();      /* save rounding direction */

result = x + y;
if (rounddir == FE_TONEAREST)
   printf("The result was rounded to the nearest value.\n");
else if (rounddir == FE_UPWARD)
   printf("The result was rounded upward.\n");
else if (rounddir == FE_DOWNWARD)
   printf("The result was rounded downward.\n");
else if (rounddir == FE_TOWARDZERO)
   printf("The result was rounded toward zero.\n");
```

fesetround

You can use the fesetround function to change the rounding direction.

```
int fesetround (int round);
```

round One of the four rounding direction constants (see Table 8-1).

DESCRIPTION

The fesetround function sets the rounding direction to the mode specified by its argument. If the value of round does not match any of the rounding direction constants, the function returns 0 and does not change the rounding direction.

By convention, if you change the rounding direction inside a function, first save the rounding direction of the calling function using fegetround and restore the saved direction at the end of the function. This way, the function does not affect the rounding direction of its caller. If the function is to be reentrant, then storage for the caller’s rounding direction must be local.

One reason to change the rounding direction would be to put bounds on errors (at least for the basic arithmetic operations and square root). Suppose you want to evaluate an expression such as

$$x = (a \times b + c \times d)/(f + g)$$

where *a*, *b*, *c*, *d*, *f*, and *g* are positive.

To make sure that the result is always larger than the exact value, you can change the expression such that all roundings cause errors in the same direction. The example that follows changes the rounding direction to compute an upper bound for the expression, and then restores the previous rounding.

EXAMPLES

```
double_t big_divide(void)
{
   double_t x_up, a, b, c, d, f, g;
   int r;               /* specifies rounding direction */

   r = fegetround();    /* save caller’s rounding direction */
   fesetround(FE_DOWNWARD);
                        /* downward rounding for denominator */
   x_up = f + g;
   fesetround(FE_UPWARD);
                        /* upward rounding for expression */
   x_up = (a * b + c * d) / x_up;
   fesetround(r);
                        /* restore caller’s rounding direction */
   return(x_up);
}
```

Controlling the Exception Flags

In MathLib, the following functions control the floating-point exception flags:

`feclearexcept`	Clears one or more exceptions.
`fegetexcept`	Saves one or more exception flags.
`feraiseexcept`	Raises one or more exceptions.
`fesetexcept`	Restores the state of one or more exception flags.
`fetestexcept`	Returns the value of one or more exception flags.

The five floating-point exception flags are defined as the constants shown in Table 8-2.

Table 8-2 Floating-point exception flags in MathLib

Exception	Constant
Inexact	`FE_INEXACT`
Divide-by-zero	`FE_DIVBYZERO`
Underflow	`FE_UNDERFLOW`
Overflow	`FE_OVERFLOW`
Invalid	`FE_INVALID`

MathLib also defines another constant, `FE_ALL_EXCEPT`, which is the logical OR of all five exceptions. Using `FE_ALL_EXCEPT`, you can manipulate all five floating-point exception flags as a single entity. The type `fexcept_t` also exists so that all the exception flags may be accessed at once.

feclearexcept

You can use the `feclearexcept` function to clear one or more floating-point exceptions.

```
void feclearexcept (int excepts);
```

`excepts` A mask indicating which floating-point exception flags should be cleared.

DESCRIPTION

The `feclearexcept` function clears the floating-point exceptions specified by its argument. The argument may be one of the constants in Table 8-2, two or more of these constants ORed together, or the constant `FE_ALL_EXCEPT`.

EXAMPLES

```
feclearexcept(FE_INEXACT);          /* clears the inexact flag */
feclearexcept(FE_INEXACT|FE_UNDERFLOW);
                     /* clears the inexact and underflow flags */
feclearexcept(FE_ALL_EXCEPT);            /* clears all flags */
```

fegetexcept

You can use the `fegetexcept` function to save the current value of one or more floating-point exception flags.

```
void fegetexcept (fexcept_t *flagp, int excepts);
```

`flagp`	A pointer to where the exception flag values are to be stored.
`excepts`	A mask indicating which exception flags to save.

DESCRIPTION

The `fegetexcept` function saves the values of the floating-point exception flags specified by the argument `excepts` to the area pointed to by the argument `flagp`. The `excepts` argument may be one of the constants in Table 8-2 on page 8-6, two or more of these constants ORed together, or the constant `FE_ALL_EXCEPT`.

EXAMPLES

```
fegetexcept(flagp, FE_INVALID);     /* saves the invalid flag */
fegetexcept(flagp, FE_INVALID|FE_OVERFLOW|FE_DIVBYZERO);
      /* saves the invalid, overflow, and divide-by-zero flags */
fegetexcept(flagp, FE_ALL_EXCEPT);        /* saves all flags */
```

feraiseexcept

You can use the `feraiseexcept` function to raise one or more floating-point exceptions.

```
void feraiseexcept (int excepts);
```

`excepts`	A mask indicating which floating-point exception flags should be set.

DESCRIPTION

The `feraiseexcept` function sets the floating-point exception flags specified by its argument. The argument may be one of the constants in Table 8-2 on page 8-6, two or more of these constants ORed together, or the constant `FE_ALL_EXCEPT`.

EXAMPLES

```
feraiseexcept(FE_OVERFLOW);         /* sets the overflow flag */
feraiseexcept(FE_INEXACT|FE_UNDERFLOW);
                        /* sets the inexact and underflow flags */
feraiseexcept(FE_ALL_EXCEPT);                /* sets all flags */
```

fesetexcept

You can use the `fesetexcept` function to restore the values of the floating-point exception flags previously saved by a call to `fegetexcept`.

```
void fesetexcept (const fexcept_t *flagp, int excepts);
```

`flagp` A pointer to the values the floating-point exception flags should have.

`excepts` A mask indicating which exception flags should have their values changed.

DESCRIPTION

The `fesetexcept` function sets the floating-point exception flags indicated by the argument `excepts` to the values indicated by the argument `flagp`. The `excepts` argument may be one of the constants in Table 8-2 on page 8-6, two or more of these constants ORed together, or the constant `FE_ALL_EXCEPT`.

You must call `fegetexcept` before this function to set the `flagp` argument. This argument cannot be set in any other way.

EXAMPLES

```
fesetexcept(flagp, FE_INVALID);  /* restores the invalid flag */
fesetexcept(flagp, FE_INVALID|FE_OVERFLOW|FE_DIVBYZERO);
   /* restores the invalid, overflow, and divide-by-zero flags */
fesetexcept(flagp, FE_ALL_EXCEPT);       /* restores all flags */
```

fetestexcept

You can use the `fetestexcept` function to find out if one or more floating-point exceptions has occurred.

```
int fetestexcept (int excepts);
```

`excepts` A mask indicating which floating-point exception flags should be tested.

DESCRIPTION

The fetestexcept function tests the floating-point exception flags specified by its argument. The argument may be one of the constants in Table 8-2 on page 8-6, two or more of these constants ORed together, or the constant FE_ALL_EXCEPT.

If all exception flags being tested are clear, fetestexcept returns a 0. If one of the flags being tested is set, fetestexcept returns the constant associated with that flag. If more than one flag is set, fetestexcept returns the result of ORing their constants together. For example, if the inexact exception is set, fetestexcept returns FE_INEXACT. If both the inexact and overflow exceptions flags are set, fetestexcept returns FE_INEXACT | FE_OVERFLOW.

EXAMPLES

```
feraiseexcept(FE_DIVBYZERO|FE_OVERFLOW);
feclearexcept(FE_INEXACT|FE_UNDERFLOW|FE_INVALID);

/* Now the divide-by-zero and overflow flags are 1, and the
   rest of the flags are 0. */

i = fetestexcept(FE_INEXACT);
                              /* i = 0 because inexact is clear */
i = fetestexcept(FE_DIVBYZERO);
                              /* i = FE_DIVBYZERO */
i = fetestexcept(FE_UNDERFLOW);
                              /* i = 0 */
i = fetestexcept(FE_OVERFLOW);
                              /* i = FE_OVERFLOW */
i = fetestexcept(FE_ALL_EXCEPT);
                              /* i = FE_DIVBYZERO | FE_OVERFLOW */
i = fetestexcept(FE_INVALID | FE_DIVBYZERO);
                              /* i = FE_DIVBYZERO */
```

Accessing the Floating-Point Environment

MathLib defines four functions that access the entire floating-point environment:

fegetenv	Returns the current environment.
feholdexcept	Saves the previous environment and clears all exception flags.
fesetenv	Sets new environmental values.
feupdateenv	Restores a previously saved environment.

These functions take parameters of type fenv_t. Type fenv_t is the environment word type. In general, the environmental access functions either take a pointer to a variable of type fenv_t or accept the macro FE_DFL_ENV, which defines the default environment (default rounding direction and all exceptions cleared).

fegetenv

You can use the fegetenv function to save the current state of the floating-point environment.

```
void fegetenv (fenv_t *envp);
```

envp   A pointer to an environment word that will store the current state of the environment upon the function's return.

DESCRIPTION

The fegetenv function saves the current state of the rounding direction modes and the floating-point exception flags in the object pointed to by its envp argument.

EXAMPLES

```
double_t func (double_t x, double_t y)
{
   fenv_t *env;

   x = x + y;        /* floating-point op; may raise exceptions */
   fegetenv(env);    /* save state of env after add */

   y = y * x;        /* floating-point op; may raise exceptions */
   .
   .
   .
}
```

feholdexcept

You can use the feholdexcept function to save the current floating-point environment and then clear all exception flags.

```
int feholdexcept (fenv_t *envp);
```

`envp` A pointer to an environment word where the environment should be saved.

DESCRIPTION

The `feholdexcept` function stores the current environment in the argument `envp` and clears the floating-point exception flags. Note that this function does not affect the rounding direction. It is the same as performing the following two calls:

```
fegetenv(envp);
feclearexcept(FE_ALL_EXCEPT);
```

Call `feholdexcept` at the beginning of a function so that the function can start with all exceptions cleared but not change the caller's environment. Use `feupdateenv` to restore the caller's environment at the end of the function. The `feupdateenv` function keeps any exceptions raised by the current function set while restoring the rest of the caller's environment. Thus, using `feholdexcept` and `feupdateenv` together preserves all raised floating-point exceptions while allowing new ones to be raised as well.

EXAMPLES

```
void subroutine(void)
{
   fenv_t *e;        /* local storage for environment */

   feholdexcept(e);  /* save caller's environment and
                        clear exceptions */

   /* subroutine's operations here */

  feupdateenv(e);    /* restore caller's environment */
}
```

fesetenv

You can use the `fesetenv` function to restore the floating-point environment.

```
void fesetenv (const fenv_t *envp);
```

`envp` A pointer to a word containing the value to which the environment should be set.

DESCRIPTION

The `fesetenv` function sets the floating-point environment to the value pointed to by its argument `envp`. The value of `envp` must come from a call to either `fegetenv` or `feholdexcept`, or it may be the constant `FE_DFL_ENV`, which specifies the default environment. In the default environment, all exception flags are clear and the rounding direction is set to the default.

EXAMPLES

```
double_t func (double_t x, double_t y)
{
   fenv_t *env;

   fesetenv(FE_DFL_ENV);      /* clear environment */

   x = x + y;        /* floating-point op; may raise exceptions */
   fegetenv(env);    /* save state of env after add */

   y = y * x;        /* floating-point op; may raise exceptions */
   fesetenv(env);    /* ignore environmental changes by
                        multiplication operator */
   .
   .
   .
}
```

feupdateenv

You can use the `feupdateenv` function to restore the floating-point environment previously saved with `feholdexcept`.

```
void feupdateenv (const fenv_t *envp);
```

`envp` A pointer to the word containing the environment to be restored.

DESCRIPTION

The `feupdateenv` function, which takes a saved environment as argument, does the following:

1. It temporarily saves the exception flags (raised by the current function).
2. It restores the environment received as an argument.
3. It signals the temporarily saved exceptions.

The feupdateenv function facilitates writing subroutines that appear to their callers to be **atomic operations** (such as addition, square root, and others). Atomic operations pass extra information back to their callers by signaling exceptions; however, they hide internal exceptions, which might be irrelevant or misleading. Thus, exceptions signaled between the feholdexcept and feupdateenv functions are hidden from the calling function unless the exceptions remain raised when the feupdateenv procedure is called.

EXAMPLES

```
/* NumFcn signals underflow if its result is denormalized,
overflow if its result is INFINITY, and inexact always, but hides
spurious exceptions occurring from internal computations. */

long double NumFcn(void)
{
   fenv_t e;                     /* local environment storage */
   enum NumKind c;               /* for class inquiry */
   fexcept_t * flagp;
   long double result;

   feholdexcept(&e);             /* save caller’s environment and
                                    clear exceptions */

      /* internal computation */

   c = fpclassify(result);       /* class inquiry */

   feclearexcept(FE_ALL_EXCEPT); /* clear all exceptions */
   feraiseexcept(FE_INEXACT);    /* signal inexact */

   if (c == FP_INFINITE)
      feraiseexcept(FE_OVERFLOW);
   else if (c == FP_SUBNORMAL)
      feraiseexcept(FE_UNDERFLOW);

   feupdateenv(&e);
   /* restore caller’s environment, and then signal
      exceptions raised by NumFcn */

   return(result);
}
```

Environmental Controls Summary

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

C Summary

Constants

Rounding Direction Modes

```
#define  FE_TONEAREST      0x00000000
#define  FE_TOWARDZERO     0x00000001
#define  FE_UPWARD         0x00000002
#define  FE_DOWNWARD       0x00000003
```

Floating-Point Exception Flags

```
#define  FE_INEXACT        0x02000000     /* inexact */
#define  FE_DIVBYZERO      0x04000000     /* divide-by-zero */
#define  FE_UNDERFLOW      0x08000000     /* underflow */
#define  FE_OVERFLOW       0x10000000     /* overflow */
#define  FE_INVALID        0x20000000     /* invalid */

#define  FE_ALL_EXCEPT     (  FE_INEXACT | FE_DIVBYZERO | FE_UNDERFLOW | \
                              FE_OVERFLOW | FE_INVALID )

#define  FE_DFL_ENV        &_FE_DFL_ENV   /* pointer to default environment*/
```

Data Types

```
typedef     long int    fenv_t;

typedef     long int    fexcept_t;
```

Environment Access Routines

Controlling the Rounding Direction

```
int fegetround              (void);
int fesetround              (int round);
```

Controlling the Exception Flags

```
void feclearexcept          (int excepts);
void fegetexcept            (fexcept_t *flagp, int excepts);
void feraiseexcept          (int excepts);
void fesetexcept            (const fexcept_t *flagp, int excepts);
int fetestexcept            (int excepts);
```

Accessing the Floating-Point Environment

```
void fegetenv               (fenv_t *envp);
int feholdexcept            (fenv_t *envp);
void fesetenv               (const fenv_t *envp);
void feupdateenv            (const fenv_t *envp);
```