Floating-point environment <fenv.h>

WG14/N468 X3J11/95-069 (Draft 8/23/95)

Jim Thomas
Taligent, Inc.
10201 N. DeAnza Blvd.
Cupertino, CA 95014-2233
jim_thomas@taligent.com

7.x Floating-point environment <fenv.h>

The header **<fenv.h>** declares two types and several macros and functions to provide access to the floating-point environment. The *floating-point environment* refers collectively to any floating-point status flags and control modes supported by the implementation.[1] A *floating-point status flag* is a system variables signifying some occurrence in the floating-point arithmetic. A *floating-point control mode* is a system variable that affects floating-point arithmetic.

[The floating-point environment as defined here includes only execution-time modes, not the myriad of possible translation-time options that may affect a program's results. Examples of such translation-time options include: chopped or rounded multiplication on CRAY Y-MP systems, D or G format for VAX, and fast or correctly-rounded divide on the Intel 860. Each option's implementation-defined or deviant properties, relative to this specification, should be well documented.]

Certain programming conventions support the intended model of use for the floating-point environment[2]:

— A function call must not alter its caller's modes, clear its caller's flags, nor depend on the state of its caller's flags unless the function is so documented.

— A function call is assumed to require default modes, unless its documentation promises otherwise or unless the function is known not to use floating-point.

— A function call is assumed to have the potential of raising floating-point exceptions, unless its documentation promises otherwise, or unless the function is known not to use floating-point.

[Libraries are encouraged to document their use, or non-use, of floating-point and their raising of floating-point exceptions.

Names in this header consistently include an `FE_` or `fe` prefix and employ certain abbreviations. The prefix calls attention to environmental access functions, which require

---

[1] This header is designed to support the exception status flags and directed-rounding control modes required by ANSI/IEEE 754 (AEC 559), and other similar floating-point state. Also it is designed to facilitate code portability among all systems. Implementation is trivial for systems that do not provide access to floating-point state.

[2] With these conventions, a programmer can safely assume default modes (or be unaware of them). Responsibilities, and (ordinarily modest) overhead, associated with accessing the floating-point environment fall on the programmer or program that does so explicitly.

an enabling `fenv_access` macro. The abbreviations `env` and `except` are used in Standard C and UNIX System V, respectively.

The interface described here is not intended to support floating-point trap handlers.]

The typedef

`fenv_t`

is a type for representing the entire floating-point environment.

The typedef

`fexcept_t`

is a type for representing the floating-point exception flags collectively, including any status the implementation associates with the flags.

Each macro

```
FE_INEXACT
FE_DIVBYZERO
FE_UNDERFLOW
FE_OVERFLOW
FE_INVALID
```

is defined if and only if the implementation supports the exception by means of the functions in 7.x.1.[3] The defined macros expand to `int` constant expressions whose values are distinct powers of 2.

The macro

`FE_ALL_EXCEPT`

is simply the bitwise OR of all exception macros defined by the implementation.

Each macro

```
FE_TONEAREST
FE_UPWARD
FE_DOWNWARD
FE_TOWARDZERO
```

is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the `fegetround` and `fesetround` functions. The defined

---

3 Unsupported macros are not defined in order to assure their use results in a translation error. A program might explicitly define such macros, to allow translation of code (perhaps never executed) containing the macros. The program could define the unsupported exception macros to be 0, for example

```
#ifndef FE_INEXACT
#define FE_INEXACT 0
#endif
```

so that a bitwise OR of macros has a reasonable effect.

554

macros expand to `int` constant expressions whose values are distinct nonnegative values.[4]

The macro

```
FE_DFL_ENV
```

represents the default floating-point environment—the one installed at program startup—and has type *pointer to* `fenv_t`. It can be used as an argument to `<fenv.h>` functions that manage the floating-point environment.

The macros

```
fenv_access_on
fenv_access_off
fenv_access_default
```

provide a means to inform the implementation when a program might access the floating-point environment to test flags or run under non-default modes.[5] Each macro can occur outside external declarations and takes effect from its occurrence until another `fenv_access` macro is encountered, or until the end of the translation unit. The effect of one of these macros appearing inside an external declaration is undefined. If part of a program tests flags or runs under non-default mode settings, but the state for the `fenv_access` macros is not *on*, then the behavior of that program is undefined. The default state (*on* or *off*) for the macros is implementation-defined.

[Previous versions of this specifications used pragmas instead of macros for this mechanism. Macros were preferred because of general limitations with pragmas and because of the wish not to require standard pragmas.]

Example

```
#include <fenv.h>
fenv_access_on
void f(double x)
{
    void g(double);
    void h(double);
    /*...*/
    g(x + 1);
    h(x + 1);
    /*...*/
}
```

If the function `g` might depend on status flags set as a side effect of the first `x + 1`, or if the second `x + 1` might depend on control modes set as a side effect of the function call

4 The rounding direction macros might expand to constants corresponding to the values of `FLT_ROUNDS`, the inquiry for the rounding direction of addition, but need not.

5 The purpose of the `fenv_access` macros is to allow certain optimizations, for example *global common subexpression elimination*, *code motion*, and *constant folding*, that could subvert flag tests and mode changes. In general, if the state of `fenv_access` is *off* then the translator can assume that default modes are in effect and the flags are not tested.

g, then this specification says the program must contain an appropriately placed invocation to `fenv_access_on`. [6]

[The performance of code under the effect of an enabling `fenv_access` macro may well be important; in fact, an algorithm may access the floating-point environment specifically for performance. The implementation should optimize as aggressively as the `fenv_access` macros allow. (See §X.9.)

An implementation could simply honor the floating-point environment in all cases and ignore the `fenv_access` macros.

Dynamic modes are potentially problematic because

1. the programmer may have to defend against undesirable mode settings—which imposes intellectual, as well as time and space, overhead.

2. the translator may not know which mode settings will be in effect or which functions change them at execution time—which inhibits optimization.

This specification attempts to address these problems without changing the dynamic nature of the modes.

An alternate approach would have been to present a model of *static modes*, with explicit utterances to the translator about what mode settings would be in effect. This would have avoided any uncertainty due to the global nature of dynamic modes or the dependency on unenforced conventions. However, some essentially dynamic mechanism still would have been needed in order to allow functions to inherit (honor) their caller's modes. The IEEE standard requires dynamic rounding direction modes. For the many architectures that maintain these modes in control registers, implementation of the static model would be more costly. Also, Standard C has no facility, other than macros and pragmas, for supporting static modes.

An implementation on an architecture that provides only static control of modes, for example through opword encodings, still could support the dynamic model, by generating multiple code streams with tests of a private global variable containing the mode setting. Only modules under an enabling `fenv_access` macro would need such special treatment. To further limit the problem, the implementation might employ additional translation options specifically to indicate where non-default modes would be admissible.]

7.x.1 Exceptions

The following functions provide access to the exception flags.[7] The `int` input argument for the functions represents a subset of floating-point exceptions, and can be constructed by bitwise ORs of the exception macros, for example `FE_OVERFLOW | FE_INEXACT`. For other argument values the behavior of these functions is undefined.

[In previous drafts of this specification, several of the exception functions returned an `int` indicating whether the `excepts` argument represented supported exceptions. This facility

---

[6] The side effects impose a temporal ordering that requires two evaluations of `x + 1`. On the other hand, without the `fenv_access_on` macro, and assuming the default state is *off*, just one evaluation of `x + 1` would suffice.

[7] The functions `fetestexcept`, `feraiseexcept`, and `feclearexcept` support the basic abstraction of flags that are either set or clear. An implementation may endow exception flags with more information—for example, the address of the code which first raised the exception; the functions `fegetexcept` and `fesetexcept` deal with the full content of flags.

555

was deemed unnecessary because `excepts & ~FE_ALL_EXCEPT` can be used to test invalidity of the `excepts` argument.]

7.x.1.1 The `feclearexcept` function

Synopsis

```
#include <fenv.h>
void feclearexcept(int excepts);
```

Description

The `feclearexcept` function clears the supported exceptions represented by its argument. The argument `excepts` represents exceptions as a bitwise OR of exception macros.

7.x.1.2 The `fegetexcept` function

Synopsis

```
#include <fenv.h>
void fegetexcept(fexcept_t *flagp, int excepts);
```

Description

The `fegetexcept` function stores an implementation-defined representation of the exception flags indicated by the argument `excepts` through the pointer argument `flagp`.

7.x.1.3 The `feraiseexcept` function

Synopsis

```
#include <fenv.h>
void feraiseexcept(int excepts);
```

Description

The `feraiseexcept` function raises the supported exceptions represented by its argument.[8] The argument `excepts` represents exceptions as a bitwise OR of exception macros. The order in which these exceptions are raised is unspecified, except as stated in X.7.6.

[The function is not restricted to accept only IEEE valid coincident expressions for atomic operations, so that the function can be used to raise exceptions accrued over several operations.]

7.x.1.4 The `fesetexcept` function

Synopsis

```
#include <fenv.h>
void fesetexcept(const fexcept_t *flagp, int excepts);
```

---

8 The effect is intended to be similar to that of exceptions raised by arithmetic operations. Hence, enabled traps for exceptions raised by this function are taken. The specification in X.7.6 is in the same spirit.

5577

Description

The `fesetexcept` function sets the complete status for those exception flags indicated by the argument `excepts`, according to the representation in the object pointed to by `flagp`. The value of `*flagp` must have been set by a previous call to `fegetexcept`; if not, the effect on the indicated exception flags is undefined. This function does not *raise* exceptions, but only sets the state of the flags.

7.x.1.5 The `fetestexcept` function

Synopsis

```
#include <fenv.h>
int fetestexcept(int excepts);
```

Description

The `fetestexcept` function determines which of a specified subset of the exception flags are currently set. The `excepts` argument specifies—as a bitwise OR of the exception macros—the exception flags to be queried.[9]

[The argument is a *mask* because querying all flags may be more expensive on some architectures.]

Returns

The `fetestexcept` function returns the bitwise OR of the exception macros corresponding to the currently set exceptions included in `excepts`.

Example

Call `f` if invalid is set, `g` if overflow is set:

```
#include <fenv.h>
fenv_access_on
int set_excepts;
/*...*/
set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
if (set_excepts & FE_INVALID) f();
if (set_excepts & FE_OVERFLOW) g();
```

7.x.2 Rounding

The `fegetround` and `fesetround` functions provide control of rounding direction modes.

---

9 This mechanism allows testing several exceptions with just one function call.

558

7.x.2.1 The `fegetround` function

Synopsis

```
#include <fenv.h>
int fegetround(void);
```

Description

The **fegetround** function gets the current rounding direction.

Returns

The **fegetround** function returns the value of the rounding direction macro representing the current rounding direction.

7.x.2.2 The `fesetround` function

Synopsis

```
#include <fenv.h>
int fesetround(int round);
```

Description

The **fesetround** function establishes the rounding direction represented by its argument **round**. If the argument does not match a rounding direction macro, the rounding direction is not changed.

Returns

The **fesetround** function returns a nonzero value if and only if the argument matches a rounding direction macro (that is, if and only if the requested rounding direction can be established).

Example

Save, set, and restore the rounding direction. Report an error and abort if setting the rounding direction fails.

```
#include <fenv.h>
#include <assert.h>
fenv_access_on
int save_round;
int setround_ok;
save_round = fegetround();
setround_ok = fesetround(FE_UPWARD);
assert(setround_ok);
/*...*/
fesetround(save_round);
```

7.x.3 Environment

The functions in this section manage the floating-point environment—status flags and control modes—as one entity.

7.x.3.1 The **fegetenv** function

Synopsis

```
#include <fenv.h>
void fegetenv(fenv_t *envp);
```

Description

The **fegetenv** function stores the current floating-point environment in the object pointed to by **envp**.

7.x.3.2 The **feholdexcept** function

Synopsis

```
#include <fenv.h>
int feholdexcept(fenv_t *envp);
```

Description

The **feholdexcept** function saves the current environment in the object pointed to by **envp**, clears the exception flags, and installs a *non-stop* (continue on exceptions) mode, if available, for all exceptions.[10]

Returns

The **feholdexcept** function returns nonzero if and only if non-stop exception handling was successfully installed.

[More appropriate for the user model prescribed in 7.x, **feholdexcept** supersedes **feprocentry** which was equivalent to

```
fegetenv(envp);
fesetenv(FE_DFL_ENV);
```

]

7.x.3.3 The **fesetenv** function

Synopsis

```
#include <fenv.h>
void fesetenv(const fenv_t *envp);
```

---

10 ANSI/EEE 754 (IEC 559) systems have a default non-stop mode, and typically at least one other mode for trap handling or aborting; if the system provides only the non-stop mode then installing it is trivial. For such systems, the **feholdexcept** function can be used in conjunction with the **feupdateenv** function to write routines that hide spurious exceptions from their callers.

Description

The **fesetenv** function establishes the floating-point environment represented by the object pointed to by **envp**. The argument **envp** must point to an object set by a call to **fegetenv**, or equal the macro **FE_DFL_ENV** or an implementation-defined value of type *pointer to* **fenv_t**. Note that **fesetenv** merely installs the state of the exception flags represented through its argument, and does not raise these exceptions.

7.x.3.4 The feupdateenv function

Synopsis

```
#include <fenv.h>
void feupdateenv(const fenv_t *envp);
```

Description

The **feupdateenv** function saves the current exceptions in its automatic storage, installs the environment represented through **envp**, and then raises (actually re-raises) the saved exceptions.

[feupdateenv was called feprocexit in earlier drafts of this specification.]

Example

Hide spurious underflow exceptions:

```
#include <fenv.h>
fenv_access_on
double f(double x)
{
   double result;
   fenv_t save_env;
   feholdexcept(&save_env);
   /*compute result*/
   if (/*test spurious underflow*/) feclearexcept(FE_UNDERFLOW);
   feupdateenv(&save_env);
   return result;
}
```