SANE Versus PowerPC Numerics

This appendix describes how PowerPC Numerics differs from the Standard Apple Numerics Environment (SANE) and tells you how to port programs that use SANE features so that they use PowerPC Numerics features instead. SANE is the numerics environment used on 680x0-based Macintosh computers. If you have written programs that perform floating-point computations for a 680x0-based Macintosh computer, that program uses SANE features. Unlike PowerPC Numerics, SANE is not compliant with the recommendations in the FPCE technical report. Compliance with the FPCE report allows a higher level of portability.

If you run a 680x0 application on a PowerPC processor-based Macintosh computer, it uses SANE instead of the PowerPC Numerics environment unless you recompile the program with a PowerPC compiler. If you want to recompile a program written for the 680x0-based Macintosh computer, you might have to modify some of your code.

Read this chapter if you are familiar with SANE and you want to know how PowerPC Numerics compares with SANE. The first section lists the differences between SANE and PowerPC Numerics. The last section provides some suggestions for porting your code.

Comparison of SANE and PowerPC Numerics

This section goes chapter by chapter through Part 1 of the *Apple Numerics Manual*, second edition, and tells where the two environments are alike and where they differ.

Floating-Point Data Formats

The single and double data formats supported by PowerPC Numerics are identical to the single and double data formats supported by SANE. PowerPC Numerics adds the double-double format not supported in SANE. PowerPC Numerics does not support the SANE floating-point formats comp (**integral value**) and 80-bit (and 96-bit) extended.

Conversions

PowerPC Numerics converts any floating-point format or integer format to any other floating-point format. SANE supports only conversions to and from the extended data format because it performs all floating-point operations in extended precision.

Conversions between binary and decimal in SANE are accurate up to a certain number of decimal digits for each floating-point data format. All conversions in PowerPC Numerics except conversions to or from double-double are correctly rounded.

Expression Evaluation

SANE uses the extended data format as the minimum evaluation format for all floating-point operations. All operations are evaluated with the greatest amount of precision possible, which ensures against midexpression overflow and underflow.

PowerPC Numerics does not specify one evaluation method but strongly recommends a single or double minimum evaluation format with widest-need evaluation. This method permits all expressions to be evaluated in as wide a format as is necessary without forcing a wider format on expressions that could be done more quickly and as accurately with a narrower format.

Note that with PowerPC Numerics, you are not ensured against midexpression overflow and underflow as you are with SANE. For example, suppose you have the following expression:

```
double d1, d2, d3, result;

d1 = d2 = d3 =  1.7E308;   /* maximum number in double */
result = (d1 + d2) / d3;
```

With PowerPC Numerics, the expression `d1 + d2` will overflow the double format, thus producing +∞. Infinity divided by the variable `d3` will still be +∞, and so the variable `result` will be assigned +∞. With SANE, `d1`, `d2`, and `d3` are converted to extended format. The expression `d1 + d2` will not overflow the extended format, and so the variable `result` will be assigned the value 2.

Infinities, NaNs, and Denormalized Numbers

Infinities, NaNs, and denormalized numbers are represented and used identically in SANE and PowerPC Numerics.

Arithmetic and Comparison Operations

SANE and PowerPC Numerics support the same seven basic arithmetic operations (add, subtract, multiply, divide, square root, remainder, and round-to-integer). SANE has only one version of the remainder and round-to-integer functions. PowerPC Numerics has two versions of the remainder function and several round-to-integer functions.

Note
SANE's square root, remainder, and round-to-integer functions return type `extended` and take type `extended` as input. PowerPC Numerics uses type `double` instead. ◆

Both SANE and PowerPC Numerics support the following comparison operators: <, <=, >=, >, ==, and !=. All other comparison operators shown in Table 6-1 on page 6-4 in this book are not supported by SANE.

Environmental Controls

SANE and PowerPC Numerics support the same rounding direction modes and the same floating-point exception flags.

SANE supports dynamic rounding precision modes because it performs all operations in extended and is therefore required by IEEE to support dynamic rounding precision modes. PowerPC Numerics does not support rounding precision modes.

SANE supports halts for each of the five floating-point exceptions. PowerPC Numerics does not currently support halts, although it might in the future.

Transcendental (Elementary) Functions

In SANE, all transcendental functions are in extended format. That is, all of them take type `extended` for floating-point input and all of them return type `extended`. In PowerPC Numerics, all transcendental functions take double for floating-point input and return type `double`. Some of the functions have a version that performs the same operation in double-double precision.

SANE supports a subset of the transcendental functions that PowerPC Numerics supports. The functions not supported by SANE are

```
erf       erfc        fdim
fmax      fmin        gamma
lgamma    nearbyint   rinttol
round     roundtol    trunc
```

Of the functions supported by both SANE and PowerPC Numerics, a few are implemented differently in the two environments. See the section “Differences in Transcendental Functions” on page A-5 for details.

Porting SANE to PowerPC Numerics

If you have a program that is written to take advantage of SANE features, you might want to port it to the PowerPC processor to take advantage of the increased speed. This section provides tips on how to do so.

Perform the following steps to be sure that your program will run on both 680x0-based and PowerPC processor-based Macintosh computers:

1. Replace all uses of type `comp` with type `double` or `long int`.
2. Replace `sane.h` and `math.h` with `fp.h` and `fenv.h`.
3. Replace uses of `extended` with `double_t` or, if this is not possible, with `long double`.
4. Replace SANE-specific functions with their MathLib equivalents. SANE-specific functions include the functions listed as implemented differently in MathLib in the section "Differences in Transcendental Functions" on page A-5, all class and sign inquiry functions, and all environmental control functions.

The following sections guide you through these four steps.

Replacing Variables of Type comp

The first step in porting a SANE program is to remove uses of the data type `comp`. The type `comp` is a floating-point type with 64 bits of precision. In SANE, type `comp` is automatically converted to extended format whenever an expression is evaluated, just like every other SANE data format. In other words, `comp` is a floating-point type disguised as an integer type. In most cases you can replace type `comp` with type `double`, which provides 53 bits of precision. If your `comp` variables require greater than 53 bits of precision, you might need to write your own integer arithmetic package.

Using MathLib Instead of the SANE Library

The next step in porting a SANE program is to use the header files `fp.h` and `fenv.h`. The files `fp.h` and `fenv.h` replace `sane.h` and `math.h`. All of the transcendental functions declared in `sane.h` are now declared in `fp.h`, and most of them work exactly the same way in the two environments. If your program includes the header file `math.h` instead of `sane.h`, you should replace it with `fp.h` as well. The `fp.h` file declares all of the functions and macros declared in the ANSI header file `math.h` plus some additional ones.

Be aware of the differences in function prototypes in the files `sane.h` and `fp.h`. If your program currently uses `sane.h`, the declarations for transcendental functions look like this:

```
extended func_name (extended func_params);
```

In other words, all transcendental functions in `sane.h` are type `extended` and take type `extended` as arguments. These declarations mean that you can pass any floating-point type to a transcendental function without losing precision.

In `fp.h`, the typical transcendental function declaration has the form

```
double_t func_name (double_t func_params);
```

The `double_t` type changes definition based on which processor the program is run on. For the PowerPC processor, `double_t` is defined to be type `double`. For the 680x0 processor, `double_t` is defined to be type `extended`. Therefore, when you change from using `sane.h` to using `fp.h`, your program will compile on both the 680x0 and the PowerPC processors and there will be no change in the way your program runs on the 680x0. For more information on the `double_t` type, see "Portable Declarations" on page A-9.

In some cases, a numeric function also has a `long double` implementation in MathLib. The declarations of the `long double` implementations are in `fp.h` and have the form

`long double` *func_name*`l (long double` *func_params*`);`

See the function descriptions in Part 2 of this book to find out if a function you are using has a `long double` implementation. If it does, you should examine the types of the parameters you are passing to that function and you should examine the return values. If a function parameter or return value requires more than 53 bits of precision, you may need to use the `long double` implementation of the function when it runs on a PowerPC processor. To do this, you simply add the letter *l* to the function call.

Replacing Extended Format Variables

When changing extended variables, first change all variables that are declared as `extended` to type `double_t`. For the 680x0 processor, `double_t` is defined as `extended`. For the PowerPC processor, `double_t` is defined as `double`. Once you make this change, your program runs with no changes on the 680x0 processor but now also runs on the PowerPC processor. Next, you need to examine each `double_t` variable to see if it will overflow on the PowerPC processor. If the variable requires more than 53 bits of precision, change its declaration to `long double`.

Using MathLib Functions

As mentioned previously, PowerPC Numerics (specifically, the MathLib library) provides a superset of the functions that SANE provides. In most cases you don't need to make any changes to your existing calls to the SANE library. However, there are a few transcendental functions that have a different implementation in MathLib. Also, the names have changed for the class and sign inquiries and floating-point environmental controls.

Differences in Transcendental Functions

The following transcendental functions are implemented differently in MathLib than in the SANE library:

- The `copysign` function does not follow the IEEE standard in SANE, which reverses the order of the function's parameters. PowerPC Numerics follows the parameter order described in the IEEE standard.

- The `exp1` function in SANE is named `expm1` in PowerPC Numerics.
- The `ipower` function is replaced with the `pow` function in PowerPC Numerics.
- The `log1` function in SANE is named `log1p` in PowerPC Numerics.
- The nextafter functions in SANE are `nextfloat`, `nextdouble`, and `nextextended`. In PowerPC Numerics, they are `nexafterf`, `nextafterd`, and `nextafterl` for `float`, `double`, and `long double`, respectively.
- The `nan` function in SANE takes a character parameter, but the PowerPC Numerics `nan` function takes a character string parameter.
- The SANE `pi` function is replaced with the constant `pi`, the SANE `inf` function is replaced with the constant `INFINITY`, and the `NAN` constant remains the same.
- The `pow` function behaves differently in the two environments. For example, in SANE `pow(NAN,0)` returns a NaN, whereas in PowerPC Numerics, `pow(NAN,0)` returns a 1.
- The `remainder` function in SANE takes three parameters, the last one being a return value. The PowerPC Numerics `remainder` function takes two parameters. The `remquo` function is analogous to the SANE `remainder` function.
- The `scalb` function does not follow the IEEE standard in SANE, which reverses the order of the function's parameters. PowerPC Numerics follows the parameter order described in the IEEE standard.

Differences in Class and Sign Inquiries

The class and sign inquiry functions declared in `sane.h` are not implemented in MathLib. Instead, MathLib provides a set of macros that perform the same actions. Table A-1 shows the declarations in `sane.h` on the left and the corresponding declaration in the MathLib header file `fp.h` on the right.

Table A-1 Class and sign inquiries in SANE versus MathLib

sane.h declaration	**fp.h declaration**

`#define SNAN 0`
`#define QNAN 1`
`#define INFINITE 2`
`#define ZERONUM 3`
`#define NORMALNUM 4`
`#define DENORMALNUM 5`
`typedef short numclass;`
	`enum NumberKind {`
	`FP_SNAN = 0,`
	`FP_QNAN,`
	`FP_INFINITE,`
	`FP_ZERO,`
	`FP_NORMAL,`
	`FP_SUBNORMAL };`

`numclass classfloat (extended x);`
`numclass classdouble(extended x);`
`numclass classcomp(extended x);`
`numclass classextended(extended x);`
	`#define fp_classify(x)`*

`long signnum (extended x);`	`#define signbit(x)`

* The `fpclassify` macro returns a long integer.

Differences in Environmental Controls

MathLib's environmental control functions are declared in the header file `fenv.h`. They affect only rounding direction modes and floating-point exceptions, and they are different from the functions that perform the same tasks in the SANE library.

If the SANE program uses rounding precision modes, you must remove this code to run it on the PowerPC processor. The PowerPC processor almost always uses less precision than SANE when evaluating expressions, so this should not be a problem. See Chapter 3, "Expression Evaluation," for details.

If the SANE program uses halts, you need to replace them with your own exception handling routines.

Replace the floating-point environmental access function or macro on the left side of Table A-2 with the corresponding function or macro on the right side. If your compiler supports the environmental access switch described in Appendix D, "FPCE Recommendations for Compilers," you must turn the switch on before using any of the functions or macros from Table A-2.

Table A-2 Environmental access functions in SANE versus MathLib

sane.h declaration	**fenv.h declaration**

`#define INVALID 1`
`#define UNDERFLOW 2`
`#define OVERFLOW 4`
`#define DIVBYZERO 8`
`#define INEXACT 16`
	`#define FE_INEXACT 0x02000000`
	`#define FE_DIVBYZERO 0x04000000`
	`#define FE_UNDERFLOW 0x08000000`
	`#define FE_OVERFLOW 0x10000000`
	`#define FE_INVALID 0x20000000`

`#define IEEEDEFAULTENV`	`#define FE_DFL_ENV &_FE_DFL_ENV`
`typedef short exception;`	`typedef long int fexcept_t;`
`typedef short environment`	`typedef long int fenv_t;`

`#define TONEAREST 0`
`#define UPWARD 1`
`#define DOWNWARD 2`
`#define TOWARDZERO 3`
	`#define FE_TONEAREST 0x00000000`
	`#define FE_TOWARDZERO 0x00000001`
	`#define FE_UPWARD 0x00000002`
	`#define FE_DOWNWARD 0x00000003`

`typedef short rounddir;`	—

`void setexception(exception e, long s);`	`int fesetexcept(const fexcept_t *flagp, int excepts);`
	`int feclearexcept(int excepts);`
	`int feraiseexcept(int excepts);`

`long testexception(exception e);`	`int fetestexcept(int excepts);`
`void setround (rounddir r);`	`int fesetround(int round);`
`rounddir getround(void);`	`int fegetround(void);`

continued

Table A-2 Environmental access functions in SANE versus MathLib (continued)

sane.h declaration	fenv.h declaration
`void setenvironment(environment e);`	`void fesetenv(const fenv_t *envp);`
`void getenvironment(environment *e);`	`void fegetenv(fenv_t *envp);`
`void procentry(environment *e);`	`int feholdexcept(fenv_t *envp);`*
`void procexit(environment e);`	`void feupdateenv(const fenv_t *envp);`

* The `feholdexcept` function, although it replaces the `procentry` SANE function, affects only the exception flags. It does not affect the rounding direction.

Listing A-1 is a C code fragment that runs on both the 680x0 and PowerPC processors. It performs the `pow` function, tests for the occurrence of the inexact exception, and prints the results.

Listing A-1 Using environmental controls in SANE and PowerPC Numerics

```
   double_t x, y, result;/* double on PowerPC,extended on 680x0 */
#ifdef __SANE__         /* 680x0 processor */
   exception fp_inexact;
#else                   /* PowerPC processor */
   fexcept_t fp_inexact;
#endif

#ifdef __SANE__         /* 680x0 processor */
   setenvironment(IEEEDEFAULTENV);
#else                   /* PowerPC processor */
   fesetenv(FE_DFL_ENV);
#endif

   result = pow(x, y);

#ifdef __SANE__         /* 680x0 processor */
   fp_inexact = testexception (INEXACT);
#else                   /* PowerPC processor */
   fp_inexact = fetestexcept (FE_INEXACT);
#endif

   printf ("pow(%g,%g) = %g\t", x, y, result);
   if (fp_inexact)
      printf ("INEXACT\n");
```

Compatibility Tools in MathLib

This section describes some tools provided in MathLib that help with compatibility between two environments. The tools include type definitions that help you make efficient, portable variable declarations and macros that are defined differently on the two architectures.

Portable Declarations

MathLib defines two floating-point type definitions, `float_t` and `double_t`, in the header file `Types.h`. If you define a variable to be `float_t` or `double_t`, it means "use the most efficient floating-point type for this architecture." Table A-3 shows the definitions for `float_t` and `double_t` on PowerPC architecture compared with 680x0 architecture.

Table A-3 `float_t` and `double_t` definitions

Architecture	`float_t`	`double_t`
PowerPC	`float`	`double`
680x0	`long double`	`long double`

The PowerPC architecture is based on the IEEE double format. The most natural format for computations is double, but the architecture allows computations in single format as well. Therefore, `float_t` is defined to be `float` (single precision) and `double_t` is defined to be `double` for the PowerPC architecture. The 680x0 architecture is based on the extended format and performs all computations in extended format regardless of the type of the operands. Therefore, `float_t` and `double_t` are both `long double` (extended precision) for the 680x0 architecture.

If you declare a variable to be type `double_t` and you compile the source code as a PowerPC application, the variable is double format. If you recompile the same source code as an 680x0 application, the variable is extended format.

If your compiler is FPCE-compliant, it also supports the pragmas that allow the most efficient floating-point type to be used for function return values, parameters, and local variables. See Appendix D, "FPCE Recommendations for Compilers," for more information on these pragmas.

Macros

You might find the following macros useful to isolate 680x0-specific code from PowerPC-specific code:

Macro	Description
`__SANE__`	Defined if `sane.h` is used
`__FP__`	Defined if `fp.h` is used
`LONG_DOUBLE_SIZE`	Returns the size in bytes of `long double` on the processor on which the program is run
`DOUBLE_SIZE`	Returns the size in bytes of `double` on the processor on which the program is run
`DECIMAL_DIG`	Returns the maximum size in digits of a decimal number that can be converted to binary