or any of its optimized forms. Unlike this expression, sinh should generate a more accurate result, particularly for small arguments. The function also covers the full range of x for which the function value is representable.

sqrt **sqrt** — This function is generally *much* faster than the apparent equivalent pow(x, 0.5).

tan **tan** — This function effectively reduces its argument to a range of π radians, centered about the X–axis. Omit adding to the argument any multiple of 2*π. The function will probably do a better job than you of eliminating multiples of 2*π. Note, however, that each multiple of 2*π in the argument reduces the useful precision of the result of tan by almost three bits. For large enough arguments, the result of the function can be meaningless even though the function reports no error.

tanh **tanh** — Use this function instead of the apparent identity:

```
tanh(x) ≡ (exp(2.0 * x) - 1.0) / (exp(2.0 * x) + 1.0)
```

or any of its optimized forms. Unlike this expression, tanh should generate a more accurate result, particularly for small arguments. The function also covers the full range of x for which the function value is representable.

Implementing <math.h>

The functions in <math.h> vary widely. I discuss them in three groups:

- the seminumerical functions that manipulate the components of floating-point values, such as the exponent, integer, and fraction parts
- the trignometric and inverse trignometric functions
- the exponential, logarithmic, and special power functions

primitives Along the way, I also present several low-level primitives. These are used by all the functions declared in <math.h> to isolate dependencies on the specific representation of floating-point values. I discussed the general properties of machines covered by this particular set of primitives starting on page 127. I emphasize once again that the parametrization doesn't cover all floating-point representations used in modern computers. You may have to alter one or more of the primitives for certain computer architectures. In rarer cases, you may have to alter the higher-level functions as well.

header **<math.h>** Figure 7.1 shows the file math.h. It contains only a few surprises. One is the masking macros. You can see that several of the math functions call other functions in turn. The masking macros eliminate one function call.

macro **HUGE_VAL** Another surprise the definition of the macro HUGE_VAL. I define it as the IEEE 754 code for +Inf. To do so, I introduce the type _Dconst. It is a union that lets you initialize a data object as an array of four *unsigned shorts*, then access the data object as a *double*. (See page 65 for a similar trick.) The data object _Hugeval is one of a handful of floating-point values that are best constructed this way.

Figure 7.1: `math.h`

```
/* math.h standard header */
#ifndef _MATH
#define _MATH
        /* macros */
#define HUGE_VAL    _Hugeval._D
        /* type definitions */
typedef const union {
    unsigned short _W[4];
    double _D;
    } _Dconst;
        /* declarations */
double acos(double);
double asin(double);
double atan(double);
double atan2(double, double);
double ceil(double);
double cos(double);
double cosh(double);
double exp(double);
double fabs(double);
double floor(double);
double fmod(double, double);
double frexp(double, int *);
double ldexp(double, int);
double log(double);
double log10(double);
double modf(double, double *);
double pow(double, double);
double sin(double);
double sinh(double);
double sqrt(double);
double tan(double);
double tanh(double);
double _Asin(double, int);
double _Log(double, int);
double _Sin(double, unsigned int);
extern _Dconst _Hugeval;
        /* macro overrides */
#define acos(x) _Asin(x, 1)
#define asin(x) _Asin(x, 0)
#define cos(x)  _Sin(x, 1)
#define log(x)  _Log(x, 0)
#define log10(x)    _Log(x, 1)
#define sin(x)  _Sin(x, 0)
#endif
```
□

Figure 7.2: xvalues.c

```
/* values used by math functions -- IEEE 754 version */
#include "xmath.h"

        /* macros */
#define NBITS   (48+_DOFF)
#if _D0
#define INIT(w0)    0, 0, 0, w0
#else
#define INIT(w0)    w0, 0, 0, 0
#endif
        /* static data */
_Dconst _Hugeval = {{INIT(_DMAX<<_DOFF)}};
_Dconst _Inf = {{INIT(_DMAX<<_DOFF)}};
_Dconst _Nan = {{INIT(_DNAN)}};
_Dconst _Rteps = {{INIT((_DBIAS-NBITS/2)<<_DOFF)}};
_Dconst _Xbig = {{INIT((_DBIAS+NBITS/2)<<_DOFF)}};
```

_Hugeval _Inf Figure 7.2 shows the file **xvalues.c** that defines this handful of values. It includes a definition for **_Inf** that matches **_Hugeval**. I provide both in case you choose to alter the definition of **HUGE_VAL**. The file also defines:

- _Nan **_Nan**, the code for a generated NaN that functions return when no operand is also a NaN
- _Rteps **_Rteps**, the square root of **DBL_EPSILON** (approximately), used by some functions to choose between different approximations
- _Xbig **_Xbig**, the inverse of **_Rteps._D**, used by some functions to choose between different approximations

The need for the last two values will become clearer when you see how functions use them.

header <yvals.h> The file **xvalues.c** is essentially unreadable. It is parametrized much like the file **xfloat.c**, shown on page 68. Both files make use of system-dependent parameters defined in the internal header **<yvals.h>**.

header "xmath.h" **xvalues.c** does not directly include **<yvals.h>**. Instead, it includes the internal header **"xmath.h"** that includes **<yvals.h>** in turn. All the files that implement **<math.h>** include **"xmath.h"**. Since that file contains an assortment of distractions, I show it in pieces as the need arises. You will find a complete listing of **"xmath.h"** in Figure 7.38. Here are the macros defined in **"xmath.h"** that are relevant to **xvalues.c**

```
#define _DFRAC  ((1<<_DOFF)-1)
#define _DMASK  (0x7fff&~_DFRAC)
#define _DMAX   ((1<<(15-_DOFF))-1)
#define _DNAN   (0x8000|_DMAX<<_DOFF|1<<(_DOFF-1))
```

If you can sort through this nonsense, you will observe that:

- the code for Inf has the largest-possible characteristic (**_DMAX**) with all fraction bits zero
- the code for generated NaN has the largest-possible characteristic with the most-significant fraction bit set

Figure 7.10: ldexp.c

```
/* ldexp function */
#include "xmath.h"

double (ldexp)(double x, int xexp)
    {                                   /* compute ldexp(x, xexp) */
    switch (_Dtest(&x))
        {                               /* test for special codes */
    case NAN:
        errno = EDOM;
        break;
    case INF:
        errno = ERANGE;
        break;
    case 0:
        break;
    default:                                            /* finite */
        if (0 <= _Dscale(&x, xexp))
            errno = ERANGE;
        }
    return (x);
    }                                                          □
```

Figure 7.11: xdunscal.c

```
/* _Dunscale function -- IEEE 754 version */
#include "xmath.h"

short _Dunscale(short *pex, double *px)
    {           /* separate *px to 1/2 <= |frac| < 1 and 2^*pex */
    unsigned short *ps = (unsigned short *)px;
    short xchar = (ps[_D0] & _DMASK) >> _DOFF;

    if (xchar == _DMAX)
        {                                         /* NaN or INF */
        *pex = 0;
        return (ps[_D0] & _DFRAC || ps[_D1]
            || ps[_D2] || ps[_D3] ? NAN : INF);
        }
    else if (0 < xchar || (xchar = _Dnorm(ps)) != 0)
        {                          /* finite, reduce to [1/2, 1) */
        ps[_D0] = ps[_D0] & ~_DMASK | _DBIAS << _DOFF;
        *pex = xchar - _DBIAS;
        return (FINITE);
        }
    else
        {                                               /* zero */
        *pex = 0;
        return (0);
        }
    }                                                          □
```

function ldexp Figure 7.10 shows the file ldexp.c. The function ldexp faces problems similar to frexp, only in reverse. Once it dispatches any special codes, it still has a nontrivial task to perform. It too calls on a low-level function. Let's look at the two low-level functions.

function _Dunscale Figure 7.11 shows the file xdunscal.c. It defines the function _Dunscale, which combines the actions of _Dtest and frexp in a form that is handier for several other math functions. By calling _Dunscale, the function frexp is left with little to do.

_Dunscale itself has a fairly easy job except when presented with a gradual underflow. A normalized value has a nonzero characteristic and an implicit fraction bit to the left of the most-significant fraction bit that is represented. Gradual underflow is signaled by a zero characteristic and a nonzero fraction with *no* implicit leading bit. Both these forms must be converted to a normalized fraction in the range [0.5, 1.0), accompanied by the appropriate binary exponent. The function _Dnorm, described below, handles this messy job.

function _Dscale Figure 7.12 shows the file xdscale.c that defines the function _Dscale. It too frets about special codes, because of the other ways that it can be called. Adding the *short* value xexp to the exponent of a finite *px can cause overflow, gradual underflow, or underflow. You even have to worry about integer overflow in forming the new exponent. That's why the function first computes the sum in a *long*.

Most of the complexity of the function _Dscale lies in forming a gradual underflow. The operation is essentially the reverse of _Dnorm.

function _Dnorm Figure 7.13 shows the file xdnorm.c that defines the function _Dnorm. It normalizes the fraction part of a gradual underflow and adjusts the characteristic accordingly. To improve performance, the function shifts the fraction left 16 bits at a time whenever possible. That's why it must be prepared to shift right as well as left one bit at a time. It may overshoot and be obliged to back up.

function fmod Figure 7.14 shows the file fmod.c. The function fmod is the last of the seminumerical functions declared in <math.h>. It is also the most complex. In principle, it subtracts the magnitude of y from the magnitude of x repeatedly until the remainder is smaller than the magnitude of y. In practice, that could take an astronomical amount of time, even if it could be done with any reasonable precision.

What fmod does instead is scale y by the largest possible power of two before each subtraction. That can still require dozens of iterations, but the result is reasonably precise. Note the way fmod uses _Dscale and _Dunscale to manipulate exponents. It uses _Dunscale to extract the exponents of x and y to perform a quick but coarse comparison of their magnitudes. If fmod determines that a subtraction might be possible, it uses _Dscale to scale x to approximately the right size.

Figure 7.12: `xdcsale.c` **Part 1**

```
/* _Dscale function -- IEEE 754 version */
#include "xmath.h"

short _Dscale(double *px, short xexp)
    {                     /* scale *px by 2^xexp with checking */
    long lexp;
    unsigned short *ps = (unsigned short *)px;
    short xchar = (ps[_D0] & _DMASK) >> _DOFF;

    if (xchar == _DMAX)                          /* NaN or INF */
        return (ps[_D0] & _DFRAC || ps[_D1]
            || ps[_D2] || ps[_D3] ? NAN : INF);
    else if (0 < xchar)
        ;                                            /* finite */
    else if ((xchar = _Dnorm(ps)) == 0)
        return (0);                                    /* zero */
    lexp = (long)xexp + xchar;
    if (_DMAX <= lexp)
        {                           /* overflow, return +/-INF */
        *px = ps[_D0] & _DSIGN ? -_Inf._D : _Inf._D;
        return (INF);
        }
    else if (0 < lexp)
        {                             /* finite result, repack */
        ps[_D0] = ps[_D0] & ~_DMASK | (short)lexp << _DOFF;
        return (FINITE);
        }
    else
        {                               /* denormalized, scale */
        unsigned short sign = ps[_D0] & _DSIGN;

        ps[_D0] = 1 << _DOFF | ps[_D0] & _DFRAC;
        if (lexp < -(48+_DOFF+1))
            xexp = -1;                    /* certain underflow */
        else
            {                           /* might not underflow */
            for (xexp = lexp; xexp <= -16; xexp += 16)
                {                            /* scale by words */
                ps[_D3] = ps[_D2], ps[_D2] = ps[_D1];
                ps[_D1] = ps[_D0], ps[_D0] = 0;
                }
            if ((xexp = -xexp) != 0)
                {                             /* scale by bits */
                ps[_D3] = ps[_D3] >> xexp
                    | ps[_D2] << 16 - xexp;
                ps[_D2] = ps[_D2] >> xexp
                    | ps[_D1] << 16 - xexp;
                ps[_D1] = ps[_D1] >> xexp
                    | ps[_D0] << 16 - xexp;
                ps[_D0] >>= xexp;
                }
            }
```

Continuing
xdscale.c
Part 2

```
        if (0 <= xexp && (ps[_D0] || ps[_D1]
            || ps[_D2] || ps[_D3]))
            {                                   /* denormalized */
            ps[_D0] |= sign;
            return (FINITE);
            }
        else
            {                           /* underflow, return +/-0 */
            ps[_D0] = sign, ps[_D1] = 0;
            ps[_D2] = 0, ps[_D3] = 0;
            return (0);
            }
        }
    }
```
□

Figure 7.13: xdnorm.c

```
/* _Dnorm function -- IEEE 754 version */
#include "xmath.h"

short _Dnorm(unsigned short *ps)
    {                               /* normalize double fraction */
    short xchar;
    unsigned short sign = ps[_D0] & _DSIGN;

    xchar = 0;
    if ((ps[_D0] &= _DFRAC) != 0 || ps[_D1]
        || ps[_D2] || ps[_D3])
        {                                       /* nonzero, scale */
        for (; ps[_D0] == 0; xchar -= 16)
            {                                   /* shift left by 16 */
            ps[_D0] = ps[_D1], ps[_D1] = ps[_D2];
            ps[_D2] = ps[_D3], ps[_D3] = 0;
            }
        for (; ps[_D0] < 1<<_DOFF; --xchar)
            {                                   /* shift left by 1 */
            ps[_D0] = ps[_D0] << 1 | ps[_D1] >> 15;
            ps[_D1] = ps[_D1] << 1 | ps[_D2] >> 15;
            ps[_D2] = ps[_D2] << 1 | ps[_D3] >> 15;
            ps[_D3] <<= 1;
            }
        for (; 1<<_DOFF+1 <= ps[_D0]; ++xchar)
            {                                   /* shift right by 1 */
            ps[_D3] = ps[_D3] >> 1 | ps[_D2] << 15;
            ps[_D2] = ps[_D2] >> 1 | ps[_D1] << 15;
            ps[_D1] = ps[_D1] >> 1 | ps[_D0] << 15;
            ps[_D0] >>= 1;
            }
        ps[_D0] &= _DFRAC;
        }
    ps[_D0] |= sign;
    return (xchar);
    }
```
□

function `_Sin` Now let's look at the trignometric functions. Figure 7.15 shows the file `xsin.c` that defines the function `_Sin`. It computes `sin(x)` if `qoff` is zero and `cos(x)` if `qoff` is one. Using such a "quadrant offset" for cosine avoids the loss of precision that occurs in adding π/2 to the argument instead. I developed the polynomial approximations from truncated Taylor series by "economizing" them using Chebychev polynomials. (If you don't know what that means, don't worry.)

Reducing the argument to the range [–π/4, π/4] must be done carefully. It is easy enough to determine how many times π/2 should be subtracted from the argument. That determines `quad`, the quadrant (centered on one of the four axes) in which the angle lies. You need the low-order two bits of `quad + qoff` to determine whether to compute the cosine or sine and whether to negate the result. Note the way the signed quadrant is converted to an unsigned value so that negative arguments get treated consistenly on all computer architectures.

What you'd like to do at this point is compute `quad`*π/2 to arbitrary precision. You want to subtract this value from the argument and still have full *double* precision after the most-significant bits cancel. Given the wide range that floating-point values can assume, that's a tall order. It's also a bit silly. As I discussed on page 135, the circular functions become progressively grainier the larger the magnitude of the argument. Beyond some magnitude, all values are indistinguishable from exact multiples of π/2. Some people argue that this is an error condition, but the C Standard doesn't say so. The circular functions must return some sensible value, and report no error, for all finite argument values.

macro `HUGE_RAD` I chose to split the difference. Adapting the approach used by Cody and Waite in several places, I represent π/2 to "one-and-a-half" times *double* precision. The header `"xmath.h"` defines the macro `HUGE_RAD` as:

```
#define HUGE_RAD    3.14e30
```

You can divide an argument up to this magnitude by π/2 and still get an value that you can convert to a *long* with no fear of overflow. The constant `c1` represents the most-significant bits of π/2 as a *double* whose least-significant 32 fraction bits are assuredly zero. (The constant `c2` supplies a full *double*'s worth of additional precision.)

That means you can multiply `c1` by an arbitrary *long* (converted to *double*) and get an exact result. Thus, so long as the magnitude of the argument is less than `HUGE_RAD`, you can develop the reduced argument to full *double* precision. That's what happens in the expression:

```
    g = (x - g * c1) - g * c2;
```

For arguments larger in magnitude than `HUGE_RAD`, the function simply slashes off a multiple of 2*π. Note the use of `_Dint` to isolate the integer part of a *double*. Put another way, once the argument goes around about a billion times, `sin` and `cos` suddenly stop trying so hard. I felt it was not worth the extra effort needed to extend smooth behavior to larger arguments.

Figure 7.15: `xsin.c` **Part 1**

```
/* _Sin function */
#include "xmath.h"

/* coefficients */
static const double c[8] = {
    -0.00000000011470879,
     0.000000002087712071,
    -0.000000275573192202,
     0.000024801587292937,
    -0.001388888888888893,
     0.041666666666667325,
    -0.500000000000000000,
     1.0};
static const double s[8] = {
    -0.000000000000764723,
     0.00000000160592578,
    -0.00000025052108383,
     0.000002755731921890,
    -0.000198412698412699,
     0.008333333333333372,
    -0.166666666666666667,
     1.0};
static const double c1 = {3294198.0 / 2097152.0};
static const double c2 = {3.139164786504813217e-7};
static const double twobypi = {0.63661977236758134308};
static const double twopi = {6.28318530717958647693};

double _Sin(double x, unsigned int qoff)
    {                              /* compute sin(x) or cos(x) */
    switch (_Dtest(&x))
        {
    case NAN:
        errno = EDOM;
        return (x);
    case 0:
        return (qoff ? 1.0 : 0.0);
    case INF:
        errno = EDOM;
        return (_Nan._D);
    default:                                           /* finite */
         {                                    /* compute sin/cos */
        double g;
        long quad;

        if (x < -HUGE_RAD || HUGE_RAD < x)
            {                          /* x huge, sauve qui peut */
            g = x / twopi;
            _Dint(&g, 0);
            x -= g * twopi;
            }
        g = x * twobypi;
        quad = (long)(0 < g ? g + 0.5 : g - 0.5);
        qoff += (unsigned long)quad & 0x3;
        g = (double)quad;
        g = (x - g * c1) - g * c2;
```

Continuing
xsin.c
Part 2

```
        if ((g < 0.0 ? -g : g) < _Rteps._D)
            {                   /* sin(tiny)==tiny, cos(tiny)==1 */
            if (qoff & 0x1)
                g = 1.0;                              /* cos(tiny) */
            }
        else if (qoff & 0x1)
            g = _Poly(g * g, c, 7);
        else
            g *= _Poly(g * g, s, 7);
        return (qoff & 0x2 ? -g : g);
         }
        }
    }                                                              □
```

Figure 7.16:
xpoly.c

```
/* _Poly function */
#include "xmath.h"

double _Poly(double x, const double *tab, int n)
    {                                         /* compute polynomial */
    double y;

    for (y = *tab; 0 <= --n; )
        y = y * x + *++tab;
    return (y);
    }                                                              □
```

The rest of the function `_Sin` is straightforward. If the reduced angle `g` is sufficiently small, evaluating a polynomial approximation is a waste of time. It also runs the risk of generating an underflow when computing the squared argument `g * g` if the reduced angle is *really* small. Here, "sufficiently small" occurs when `g * g` is less than `DBL_EPSILON`, defined in `<float.h>`. Note the use of the double constant `_Rteps._D` to speed this test.

`_Poly` Figure 7.16 shows the file `xpoly.c` that defines the function `_Poly`. The function `_Sin` uses `_Poly` to evaluate a polynomial by Horner's Rule.

`cos` `sin` Figure 7.17 shows the file `cos.c` and Figure 7.18 shows the file `sin.c`. These define the trivial functions `cos` and `sin`. The header `<math.h>` defines masking macros for both.

function `tan` Figure 7.19 shows the file `tan.c`. The function `tan` strongly resembles the other circular functions `sin` and `cos`. It too reduces its argument to the interval $[-\pi/4, \pi/4]$. The major difference is the way the function is approximated over this reduced interval. Because it has poles at multiples of $\pi/2$, the tangent is better approximated by a ratio of polynomials. Cody and Waite supplied the coefficients.

function `_Asin` Now consider the inverse trignometric functions. Figure 7.20 shows the file `xasin.c` that defines the function `_Asin`. It computes `asin(x)` if `qoff` is zero and `acos(x) if qoff` is one. That avoids the need to tinker twice with the result for `acos`.

Figure 7.17: cos.c

```
/* cos function */
#include <math.h>

double (cos)(double x)
    {                                                /* compute cos */
    return (_Sin(x, 1));
    }                                                              □
```

Figure 7.18: sin.c

```
/* sin function */
#include <math.h>

double (sin)(double x)
    {                                                /* compute sin */
    return (_Sin(x, 0));
    }                                                              □
```

_Asin first determines y, the magnitude of the argument. It computes the intermediate result (also in y) five different ways:

- If y < _Rteps._D, use the argument itself.
- Otherwise, if y < 0.5, use a ratio of polynomials approximation from Cody and Waite.
- Otherwise, if y < 1.0, use the same approximation to compute 2 * asin(sqrt(1 - x) / 2)) (effectively). The actual arithmetic takes pains to minimize loss of intermediate significance.
- Otherwise, if y == 1.0, use zero.
- Otherwise, y > 1.0 and the function reports a domain error.

The concern with any such piecemeal approach is introducing discontinuities at the boundaries. The most worrisome boundary in this case occurs when y equals 0.5.

_Asin determines the final result from notes taken in idx along the way:

- If idx & 1, the arccosine was requested, not the arcsine.
- If idx & 2, the argument was negated.
- If idx & 4, the magnitude of the argument was greater than 0.5.

The final fixups involve adding various multiples of $\pi/4$ and negating the works. The sums are formed in stages to prevent loss of significance.

acos asin Figure 7.21 shows the file acos.c and Figure 7.22 shows the file asin.c. These define the trivial functions acos and asin. The header <math.h> defines masking macros for both.

atan atan2 The last of the inverse trignometric functions is the arctangent. It comes in two forms, atan(x) and atan2(y, x). Both call a common function _Atan to do the actual computation. Unlike the earlier trignometric functions, however, the common function is not the best one to show first. Figure 7.23 shows the file atan.c. Figure 7.24 shows the file atan2.c. It defines the function atan2 that reveals how the three functions work together.