// Filename: FixedMath.h
//
// Copyright 2004-2016 Mitsubishi Electric Research Laboratories (MERL)
// An interface and implementation for fixed point math functions
// Eric Chan, Ronald Perry, and Sarah Frisk

//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
// (INLINE to the keyword used by their compiler to see below)
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
#!/ifdef _NTO_FIXED_MATH_
#define _NTO_FIXED_MATH_

//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
// To avoid multiple inclusion of header files
//------------------------------------------------------------------------------
 Usuario ndef _cplusplus
extern "C" {
#endif

//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
治安 MATH MODE SELECTION
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
// _NTO_MATH_MODE must be set to one of the following constants:
//
//
// - _NTO_MATH_FIXED_C_32
// - _NTO_MATH_FIXED_C_64
// - _NTO_MATH_FIXED_ASM_X86

// If _NTO_MATH_MODE is set to _NTO_MATH_FIXED_C_32, this system uses a fixed point
// implementation for all internal computations. This fixed point implementation is
// written in ANSI C and is portable across systems that support 32-bit integers and
// two's complement arithmetic.
//
// If _NTO_MATH_MODE is set to _NTO_MATH_FIXED_C_64, this system uses a fixed point
// implementation for all internal computations. This fixed point implementation is
// portable across systems that support 32-bit integers, 64-bit integers, and two's
// complement arithmetic. Applications must define _NTO_I64 appropriately (see
// below). This implementation requires that the system performs sign extension when
// the left operand of a right shift is a signed integer. In general, this
// implementation is significantly faster than _NTO_MATH_FIXED_C_32.
//
// If _NTO_MATH_MODE is set to _NTO_MATH_FIXED_ASM_X86, this system uses a fixed point
// implementation for all internal computations. This fixed point implementation is
// written in optimized x86 assembly. Applications must define _NTO_I64 appropriately
// (see below). This implementation compiles only to x86-based systems (e.g., Pentium
// systems) and is usually significantly faster than _NTO_MATH_FIXED_C_64.

//------------------------------------------------------------------------------
#define _NTO_MATH_FIXED_C_32 1
#define _NTO_MATH_FIXED_C_64 2
#define _NTO_MATH_FIXED_ASM_X86 3

//------------------------------------------------------------------------------
#define _NTO_MATH_MODE _NTO_MATH_FIXED_C_64

//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
// Applications should set _NTO_INLINE to the keyword used by their compiler to
// identify inline functions. The default setting is __inline, the keyword used by

//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
// (INLINE to the keyword used by their compiler to see below)
/**
 * the Microsoft Visual Studio 2008 compiler to identify inline functions.
 */
#define NTO_INLINE __inline

/**
 * Applications should set NTO_I64 to the keyword used by their compiler to
 * represent a 64-bit signed integer
 */
typedef long long NTO_I64;

/**
 * FUNDAMENTAL DATA TYPES
 */

/**
 * Fundamental data types for characters, Booleans, integers, and floating point
 * numbers
 */
typedef char NTO_I8;
typedef short NTO_I16;
typedef int NTO_I32;
typedef unsigned char NTO_U8;
typedef unsigned short NTO_U16;
typedef unsigned int NTO_U32;
typedef float NTO_F32;
typedef double NTO_F64;
typedef int NTO_Bool;

/**
 * TERMINOLOGY
 */

/**
 * The documentation in this file and FixedMath.c uses the following terminology.
 * In a N-bit integer, the least significant bit (i.e., the LSB) is bit 0 and the
 * most significant bit (i.e., the MSB) is bit N-1. Binary representations are
 * written with the MSB to the left and the LSB to the right. For example, the
 * binary representation of a 32-bit unsigned integer with the mathematical value 1
 * is 00000000000000000000000000000001, and the binary representation of a 32-bit
 * unsigned integer with the mathematical value 2147483648 (hexadecimal 0x80000000)
 * is 10000000000000000000000000000000.
 */

/**
 * A range of consecutive bits is denoted by M:L, where M is the most significant
 * bit of the range and L is the least significant bit. For example, bits 15:0 of a
 * 32-bit integer are the 16 least significant bits of the integer, and bits 23:8
 * are the middle 16 bits of the integer.
 */

/**
 * The most significant bits of an integer are called the "high" bits and the least
 * significant bits of an integer are called the "low" bits. For example, bits 15:0
 * of a 32-bit integer are called the low 16 bits, and bits 31:16 are called the
 * high 16 bits.
 */

/**
 * A fixed point number with I integer bits and F fractional bits is called an I.F
 * fixed point number. For example, a 32-bit fixed point number with 24 integer bits
 * and 8 fractional bits is called a 24.8 fixed point number. Fixed point numbers
 * are unsigned unless stated otherwise.
 */

/**
 * FIXED POINT DATA TYPES
 */

/**
 * An NTO_I1616 is a 32-bit two's complement signed fixed point data type with 1
 * sign bit, 15 integer bits, and 16 fractional bits. The MSB (i.e., bit 31) is the
 * sign bit, bits 30:16 are the integer bits, and bits 15:0 are the fractional bits.
 */
typedef NTO_I1616 NTO_I1616;

---

An NTO_I1616 value R represents the mathematical value (R / 65536.0).

**Example 1:**
- **NTO_I1616 hexadecimal value:** 0x00000000
- **NTO_I1616 binary representation:** 0 0000000000000000 (LSB)
- **Mathematical value:** 0

**Example 2:**
- **NTO_I1616 hexadecimal value:** 0x00010000
- **NTO_I1616 binary representation:** 0 0000000000000001 0000000000000000 (LSB)
- **Mathematical value:** 1

**Example 3:**
- **NTO_I1616 hexadecimal value:** 0xfffff000
- **NTO_I1616 binary representation:** 1 1111111111111111 0000000000000000 (LSB)
- **Mathematical value:** -1

**Example 4:**
- **NTO_I1616 hexadecimal value:** 0x0000ffff
- **NTO_I1616 binary representation:** 0 0000000000000000 1111111111111111 (LSB)
- **Mathematical value:** 65535 / 65536

**Example 5:**
- **NTO_I1616 hexadecimal value:** 0x00000001
- **NTO_I1616 binary representation:** 0 0000000000000001 0000000000000000 (LSB)
- **Mathematical value:** 1 / 65536
- **Remark:** minimum representable positive value

**Example 6:**
- **NTO_I1616 hexadecimal value:** 0xffffffff
- **NTO_I1616 binary representation:** 1 1111111111111111 1111111111111111 (LSB)
- **Mathematical value:** -1 / 65536
- **Remark:** maximum representable negative value

**Example 7:**
- **NTO_I1616 hexadecimal value:** 0x7fffffff
- **NTO_I1616 binary representation:** 0 1111111111111111 1111111111111111 (LSB)
- **Mathematical value:** 32767 + (65535 / 65536)
- **Remark:** maximum representable value

**Example 8:**
- **NTO_I1616 hexadecimal value:** 0x80000000
- **NTO_I1616 binary representation:** 1 0000000000000000 0000000000000000 (LSB)
- **Mathematical value:** -32768
- **Remark:** minimum representable value

**Example 9:**
- **NTO_I1616 hexadecimal value:** 0x0003243f
- **NTO_I1616 binary representation:** 0 0000000000000011 0010010000111111 (LSB)
- **Mathematical value:** 3.1415863037109375
- **Remark:** approximation to pi

---

typedef NTO_I32 NTO_I32;

---

An NTO_U0032 is an unsigned fixed point data type with 32 fractional bits. An NTO_U0032 value R represents the mathematical value (R / 4294967296.0) (i.e., (R / (2 ^ 32))).

**Example 1:**
- **NTO_U0032 hexadecimal value:** 0x00000000
- **NTO_U0032 binary representation:** 0000000000000000000000000000000000000000 (LSB)
- **Mathematical value:** 0
- **Remark:** minimum representable value

**Example 2:**
- **NTO_U0032 hexadecimal value:** 0x00000001
typedef NTO_U032 NTO_U0032;

An NTO_U0032 is a 32-bit two's complement signed fixed point data type with 1 sign bit, 23 integer bits, and 8 fractional bits. The MSB (i.e., bit 31) is the sign bit, bits 30:8 are the integer bits, and bits 7:0 are the fractional bits.

Example 3:
// NTO_U0032 hexadecimal value: 0xffffffff
// NTO_U0032 binary representation: 11111111111111111111111111111111 (LSB)
// Mathematical value: 4294967295 / 4294967296
// Remark: maximum representable value

Example 4:
// NTO_U0032 hexadecimal value: 0x80000000
// NTO_U0032 binary representation: 10000000000000000000000000000000 (LSB)
// Mathematical value: 0.5

Example 5:
// NTO_U2408 hexadecimal value: 0x00000001
// NTO_U2408 binary representation: 00000000000000000000000000000001 00000000 (LSB)
// Mathematical value: 1 / 256
// Remark: minimum representable positive value

Example 6:
// NTO_U2408 hexadecimal value: 0xffffffff
// NTO_U2408 binary representation: 11111111111111111111111111111111 00000000 (LSB)
// Mathematical value: 4294967295 / 256
// Remark: maximum representable negative value

Example 7:
// NTO_U2408 hexadecimal value: 0x7fffffff
// NTO_U2408 binary representation: 01111111111111111111111111111111 11111111 (LSB)
// Mathematical value: 8388607 + (255 / 256)
// Remark: maximum representable value

Example 8:
// NTO_U2408 hexadecimal value: 0x80000000
// NTO_U2408 binary representation: 10000000000000000000000000000000 (LSB)
// Mathematical value: -8388608
// Remark: minimum representable value
typedef NTO_I32 NTO_I2408;

// NTO_I1161 FIXED POINT CONSTANTS
#define NTO_I1161_CONST_1 ((NTO_I1161) 0x00010000) // 1.0000000000000000
#define NTO_I1161_CONST_ONE_HALF ((NTO_I1161) 0x00008000) // 0.5000000000000000
#define NTO_I1161_CONST_ONE_THIRD ((NTO_I1161) 0x00005555) // 0.3333333333333333
#define NTO_I1161_CONST_ONE_SIXTH ((NTO_I1161) 0x00002aaa) // 0.1666666666666666
#define NTO_I1161_CONST_ONE_FIFTH ((NTO_I1161) 0x00003333) // 0.2000000000000000
#define NTO_I1161_CONST_ONE_TENTH ((NTO_I1161) 0x0000199a) // 0.1000061035156250
#define NTO_I1161_CONST_FIVE_THIRDS ((NTO_I1161) 0x0001aaaa) // 0.6666666666666666
#define NTO_I1161_CONST_FOUR_FIFTHS ((NTO_I1161) 0x0000cccc) // 0.8000000000000000
#define NTO_I1161_CONST_EPS ((NTO_I1161) 0x000000002000) // 0.0000030515781250
#define NTO_I1161_CONST_DELTA ((NTO_I1161) 0x000000000001) // 0.000000015287890625
#define NTO_I1161_CONST_PI ((NTO_I1161) 0x0000000000000002) // 0.000000031415926535
#define NTO_I1161_CONST_TWO_PI ((NTO_I1161) 0x0000000000000000) // 0.000000062831853071
#define NTO_I1161_CONST_HALF_PI ((NTO_I1161) 0x0000000000000000) // 0.000000031415926535
#define NTO_I1161_CONST_TWO_OVER_PI ((NTO_I1161) 0x0000000000000000) // 0.000000062831853071
#define NTO_I1161_CONST_SQRT5 ((NTO_I1161) 0x0000000000000000) // 0.000000022360679774

// NTO_I2408 FIXED POINT CONSTANTS
#define NTO_I2408_CONST_32 ((NTO_I2408) 0x00002000) // 32.0000000000000000

// rcpTable[] is a table that maps an integer i to its reciprocal (i.e., 1/i). Each element of the table is stored as an NTO_I1161 fixed point value. The table contains 512 elements and is precomputed as follows:
// rcpTable[i] = ((i > 0) ? floor(65536.0 / i) : 0), where i is an integer that lies in the range [0, 511]. Note that excess fractional bits are discarded (i.e., each element is rounded towards zero).
// The first element of the table (i.e., the element that corresponds to the undefined reciprocal of zero) is arbitrarily set to zero.

static const NTO_I1161 rcpTable[] = {
    0x000000, 0x100000, 0x080000, 0x05555, 0x040000, 0x03333, 0x02aaa, 0x02492, 0x020000, 0x01c71, 0x01999, 0x01745, 0x01555, 0x013b1, 0x01249, 0x01111, 0x01000, 0x00f0f, 0x00e38, 0x00d79, 0x00ccc, 0x00c30, 0x00b2a, 0x00b21, 0x00aaa, 0x00a3d, 0x009d8, 0x0097b, 0x00924, 0x008d3, 0x00888, 0x00842, 0x00808, 0x007c1, 0x00787, 0x00750, 0x0071c, 0x006eb, 0x006bc, 0x00690, 0x00666, 0x0063e, 0x00618, 0x005f4, 0x005d1, 0x005b0, 0x00590, 0x00572, 0x00555, 0x00539, 0x0051e, 0x00505, 0x004ec, 0x004d4, 0x004bd, 0x004a7, 0x00492, 0x0047d, 0x00469, 0x00456, 0x00444, 0x00432, 0x00421, 0x00410, 0x00400, 0x003f0, 0x003e0, 0x003d2, 0x003c3, 0x003b5, 0x003a8, 0x0039b, 0x0038e, 0x00381, 0x00375, 0x00369, 0x0035e, 0x00353, 0x00348, 0x0033d, 0x00333, 0x00329, 0x0031f, 0x00315, 0x0030c, 0x00303, 0x002fa, 0x002f1, 0x002e8, 0x002e0, 0x002d8, 0x002d0, 0x002c8, 0x002c0, 0x002b9, 0x002b1, 0x002a2, 0x002a3, 0x0029c, 0x00295, 0x0028f, 0x00288, 0x00282, 0x0027c, 0x00276, 0x00270, 0x0026a, 0x00264, 0x0025e, 0x00259, 0x00253, 0x0024e, 0x00249, 0x00243, 0x0023e, 0x00239, 0x00234, 0x00230, 0x0022b, 0x00226, 0x00222, 0x0021d, 0x00219, 0x00214, 0x00210, 0x0020c, 0x00208, 0x00204,
Compute and return (f * 255), where f is an NTO_U0032 fixed point value that lies
in the mathematical range [0, 1) and the computed result is an 8-bit unsigned
integer that lies in the range [0, 255).

U0032_TO_U8() is implemented as a macro. The first part of the macro (i.e., \( f - \)
\( f >> 8 \)) treats f as a 8.24 fixed point integer and evaluates \( (f * 256 - f) = \)
\( (f * (256 - 1)) = (f * 255) \). Note that converting f to an 8.24 fixed point value
// can be implemented by shifting the right by 8 bits (i.e., \( f >> 8 \)).
// Multiplying the 8.24 fixed point value by 256 is equivalent to shifting the
// value to the left by 8 bits (i.e., \( f >> 8 << 8 \)). Therefore, converting f from
// a 0.32 fixed point value to an 8.24 fixed point value and then multiplying the
// result by 256 is simply f.
// The second part of the macro converts the 8.24 fixed point product \( (f * 255) \) to
// an 8-bit integer by shifting the product to the right by 24 bits.
// The following examples omit some of the macro's parentheses to make the examples easier to follow.

Example 1. Consider the case where \( f = 0 \):

\[
(f - (f >> 8)) >> 24 = (0 - (0 >> 8)) >> 24
\]

\[
= (0 - 0) >> 24
\]

\[
= 0 >> 24
\]

\[
= 0
\]

Example 2. Consider the case where \( f = 0xffffffff \) (i.e., \( f \) has the mathematical value \((0xffffffff / 2^{32}) = 0.9999999976716935634613037109375\)):

\[
(f - (f >> 8)) >> 24 = (0xffffffff - (0xffffffff >> 8)) >> 24
\]

\[
= (0xffffffff - 0x00000000) >> 24
\]

\[
= 0xff000000 >> 24
\]

\[
= 0xff
\]

\[
= 255
\]

Example 3. Consider the case where \( f = 0x80000000 \) (i.e., \( f \) has the mathematical value \((0x80000000 / 2^{32}) = 0.5\)):

\[
(f - (f >> 8)) >> 24 = (0x80000000 - (0x80000000 >> 8)) >> 24
\]

\[
= (0x80000000 - 0x00080000) >> 24
\]

\[
= 0x7f800000 >> 24
\]

\[
= 0x7f
\]

\[
= 127
\]

// Define U0032_TO_U8(f) \(((f) - ((f) >> 8)) >> 24\)

// Compute and return \((f * 255)\), where \( f \) is an NTO_11616 fixed point value that lies in the mathematical range \([0, 1]\) and the computed result is an 8-bit unsigned integer that lies in the range \([0, 254]\). Note that the maximum value of the computed result is 254 and not 255 because the fractional bits of the computed result are truncated (i.e., the computed result is rounded towards zero); see Example 2 below.

I1616_TO_U8() is implemented as a macro. The first part of the macro (i.e., \((f << 8) - f\)) evaluates \(((f * 256) - f) = (f * (256 - 1)) = (f * 255)\). The second part of the macro converts the 16.16 fixed point product \((f * 255)\) to an 8-bit integer by shifting the product to the right by 16 bits.

The following examples omit some of the macro's parentheses to make the examples easier to follow.

Example 1. Consider the case where \( f = 0 \):

\[
((f << 8) - f) >> 16 = ((0 << 8) - 0) >> 16
\]

\[
= (0 - 0) >> 16
\]

\[
= 0 >> 16
\]

\[
= 0
\]

Example 2. Consider the case where \( f = 0x0000ffff \) (i.e., \( f \) has the mathematical value \((0x0000ffff / 2^{16}) = 0.9999847412109375\)):

\[
((f << 8) - f) >> 16 = ((0x00000000 << 8) - 0x00000000) >> 16
\]

\[
= (0x00ffff00 - 0x00000000) >> 16
\]

\[
= 0xff01 >> 16
\]

\[
= 0xfe
\]

\[
= 254
\]

Example 3. Consider the case where \( f = 0x00008000 \) (i.e., \( f \) has the mathematical value \((0x00008000 / 2^{16}) = 0.5\)):
NTO_INLINE static NTO_I32 CountLeadingZeros (NTU U32 u) {
    //----Initialize the number of leading zeroes to 0 to prepare for incremental
    //----updates below
    NTO_I32 numLeadingZeros = 0;

    //----If the most significant 16 bits of u are clear (i.e., u < 0x00010000), then
    //-----increment numLeadingZeros by 16 and shift the least significant 16 bits of
    //-----u to the most significant 16 bits
if (u < 0x00010000) {
    numLeadingZeroes += 16;
    u <<= 16;
}

//-----If the most significant 8 bits of u are clear (i.e., u < 0x01000000), then
//-----increment numLeadingZeroes by 8 and shift the least significant 24 bits of
//-----u to the most significant 24 bits
if (u < 0x01000000) {
    numLeadingZeroes += 8;
    u <<= 8;
}

//-----If u contains any remaining 1 bits, they must be in the most significant 8
//-----bits (i.e., bits 31:24). Use a precomputed 8-bit table to determine the
//-----number of leading zeroes in these 8 bits and increment numLeadingZeroes.
numLeadingZeroes += clzTable[u >> 24];

//-----Return the number of leading zeroes
return (numLeadingZeroes);

//------------------------------------------------------------------------------
// Multiply two 32-bit unsigned integers x and y and return only the most
// significant 32 bits of the 64-bit product (i.e., bits 63:32).
// All assembly and C implementations produce bit-identical results.
// Performance notes:
// Intel Centrino Core Duo T2500 (2 MB L2, 2.0 GHz, FSB 677 MHz), MSVC 6 compiler,
// Release mode:
//  - NTO_MATH_FIXED_C_64 is ~3.0x as fast as NTO_MATH_FIXED_C_32
//  - NTO_MATH_FIXED_ASM_X86 is ~1.0x as fast as NTO_MATH_FIXED_C_64
//  - NTO_MATH_FIXED_ASM_X86 is ~3.0x as fast as NTO_MATH_FIXED_ASM_X64
//------------------------------------------------------------------------------
NTO_INLINE static NTO_U32 UMUL64_HIGH32 (NTO_U32 x, NTO_U32 y)
{
    #if (NTO_MATH_MODE == NTO_MATH_FIXED_C_64)
    {
        // C implementation (64-bit). Compute (x * y) and return the most
        // significant 32 bits of the 64-bit product.
        return (((NTO_U32)(((NTO_I64)x) * ((NTO_I64)y)) >> 32));
    }
    #elif (NTO_MATH_MODE == NTO_MATH_FIXED_C_32)
    {
        // C implementation (32-bit). Separate x into two 16-bit values.
        NTO_U32 xLow = x & 0xffff;
        NTO_U32 xHigh = x >> 16;

        // Separate y into two 16-bit values
        NTO_U32 yLow = y & 0xffff;
        NTO_U32 yHigh = y >> 16;

        // Compute partial products
        NTO_U32 zLow = xLow * yLow;
        NTO_U32 zMid1 = xLow * yHigh;
        NTO_U32 zMid2 = xHigh * yLow;
        NTO_U32 zHigh = xHigh * yHigh;
    }
/**
 * Add the middle 32-bit values. If the resulting value of zMid is less
 * than zMid2 (i.e., zMid < zMid2), then a 32-bit unsigned integer
 * overflow occurred, which means that a carry bit must be added to zHigh
 * below.
 */
 zMid1 += zMid2;

 /**
 * Compute the most significant 32 bits of the 64-bit product and add the
 * carry bit
 */
 zHigh += (((zMid1 < zMid2) << 16) + (zMid1 >> 16));

 /**
 * Move the low 16 bits of the middle sum to the high 16 bits
 */
 zMid1 <<= 16;

 /**
 * Compute the least significant 32 bits of the 64-bit product. If the
 * resulting value of zLow is less than zMid (i.e., zLow < zMid1), then a
 * 32-bit unsigned integer overflow occurred, which means that a carry bit
 * must be added to zHigh below.
 */
 zLow += zMid1;

 /**
 * Add the carry bit
 */
 zHigh += (zLow < zMid1);

 /**
 * Return the high 32 bits of the product
 */
 return(zHigh);
}

#elif (NTO_MATH_MODE == NTO_MATH_FIXED_ASM_X86)
{
  /**
   * x86 assembly implementation
   */
  __asm {

    /**
     * Get parameters x and y
     */
    mov eax, x;
    mov ebx, y;

    /**
     * [edx:eax] = 64-bit product of x and y
     */
    mul ebx;

    /**
     * Return the high 32 bits (i.e., edx) of the product
     */
    mov eax, edx;
  }
}
#endif
NTO_INLINE static NTO_U32 UMUL64_MID32 (NTO_U32 x, NTO_U32 y)
{
    #if (NTO_MATH_MODE == NTO_MATH_FIXED_C_64)
    {
        //----C implementation (64-bit). Compute (x * y) and return the middle 32
        //----bits (i.e., bits 47:16) of the 64-bit product.
        return((NTO_U32) (((((NTO_I64) x) * ((NTO_I64) y)) >> 16));
    }
    #elif (NTO_MATH_MODE == NTO_MATH_FIXED_C_32)
    {
        //----C implementation (32-bit). Separate x into two 16-bit values.
        NTO_U32 xLow = x & 0xffff;
        NTO_U32 xHigh = x >> 16;

        //----Separate y into two 16-bit values
        NTO_U32 yLow = y & 0xffff;
        NTO_U32 yHigh = y >> 16;

        //----Compute partial products
        NTO_U32 zLow = xLow * yLow;
        NTO_U32 zMid1 = xLow * yHigh;
        NTO_U32 zMid2 = xHigh * yLow;
        NTO_U32 zHigh = xHigh * yHigh;

        //----Add the middle 32-bit values. If the resulting value of zMid1 is less
        //----than zMid2 (i.e., zMid1 < zMid2), then a 32-bit unsigned integer
        //----overflow occurred, which means that a carry bit must be added to zHigh
        //----below.
        zMid1 += zMid2;

        //----Compute the most significant 32 bits of the 64-bit product and add the
        //----carry bit
        zHigh += (((zMid1 < zMid2) << 16) + (zMid1 >> 16));

        //----Move the low 16 bits of the middle sum to the high 16 bits
        zMid1 <<= 16;

        //----Compute the least significant 32 bits of the 64-bit product. If the
        //----resulting value of zLow is less than zMid1 (i.e., zLow < zMid1), then a
        //----32-bit unsigned integer overflow occurred, which means that a carry bit
        //----must be added to zHigh below.
        zLow += zMid1;

        //----Add the carry bit
        zHigh += (zLow < zMid1);

        //----Return the middle 32 bits (i.e., bits 47:16) of the product
        return((zHigh << 16) | (zLow >> 16));
    }
    #elif (NTO_MATH_MODE == NTO_MATH_FIXED_ASM_X86)
    {
        //----x86 assembly implementation
        __asm {

            //----Get parameters x and y
            mov eax, x;
            mov ebx, y;
            ...
        }
    }
}
---{edx:eax} = 64-bit product of x and y
mul ebx;

-----Extract the middle 32 bits (i.e., bits 47:16) of the 64-bit product
shl edx, 16;
shr eax, 16;
or eax, edx;
}
#endif

// Multiply two 32-bit unsigned integers x and y and return the 64-bit product in
// two 32-bit unsigned integers. On output, zHighOut contains the high 32 bits of
// the product and zLowOut contains the low 32 bits of the product.
// All assembly and C implementations produce bit-identical results.
// Performance notes:
// Intel Centrino Core Duo T2500 (2 MB L2, 2.0 GHz, FSB 677 MHz), MSVC 6 compiler,
// Release mode:
// - NTO_MATH_FIXED_C_64 is ~2.9x as fast as NTO_MATH_FIXED_C_32
//INLINE
C:\Users\perry\Desktop\MiniNitro\Nitro\FixedMath.h
static void UMUL64 (NTO_U32 x, NTO_U32 y, NTO_U32 *zHighOut, NTO_U32 *zLowOut)
{
    #if ((NTO_MATH_MODE == NTO_MATH_FIXED_C_64) ||
         (NTO_MATH_MODE == NTO_MATH_FIXED_ASM_X86))
    {
        //C implementation (64-bit)
        NTO_I64 z64 = (((NTO_I64) x) * ((NTO_I64) y));
        *zLowOut = (NTO_U32) (z64);
        *zHighOut = (NTO_U32) (z64 >> 32);
    }
    #elif (NTO_MATH_MODE == NTO_MATH_FIXED_C_32)
    {
        //C implementation (32-bit). Separate x into two 16-bit values.
        NTO_U32 xLow = x & 0xffff;
        NTO_U32 xHigh = x >> 16;

        //Separate y into two 16-bit values
        NTO_U32 yLow = y & 0xffff;
        NTO_U32 yHigh = y >> 16;

        //Compute partial products
        NTO_U32 zLow = xLow * yLow;
        NTO_U32 zMid1 = xLow * yHigh;
        NTO_U32 zMid2 = xHigh * yLow;
        NTO_U32 zHigh = xHigh * yHigh;

        //Add the middle 32-bit values. If the resulting value of zMid1 is less
        //than zMid2 (i.e., zMid1 < zMid2), then a 32-bit unsigned integer
        //overflow occurred, which means that a carry bit must be added to zHigh
        //below.
        zMid1 += zMid2;

        //Compute the most significant 32 bits of the 64-bit product and add the
        //carry bit
zHigh += (((zMid1 < zMid2) << 16) + (zMid1 >> 16));

  //----Move the low 16 bits of the middle sum to the high 16 bits
zMid1 <<= 16;

  //----Compute the least significant 32 bits of the 64-bit product. If the
  //----resulting value of zLow is less than zMid1 (i.e., zLow < zMid1), then a
  //----32-bit unsigned integer overflow occurred, which means that a carry bit
  //----must be added to zHigh below.
zLow += zMid1;

  //----Add the carry bit
zHigh += (zLow << zMid1);

  //----Store the product
  *zLowOut = zLow;
  *zHighOut = zHigh;
}
#endif
}

//-----------------------------------------------------------------------------------
// Return 1 if a is less than b, where a and b are NTO_I1616 fixed point values;
// return zero otherwise
//-----------------------------------------------------------------------------------
NTO_INLINE static NTO_I32 I1616_LT (NTO_I1616 a, NTO_I1616 b) {
  return (a < b);
}

//-----------------------------------------------------------------------------------
// Return 1 if a is less than or equal to b, where a and b are NTO_I1616 fixed
// point values; return zero otherwise
//-----------------------------------------------------------------------------------
NTO_INLINE static NTO_I32 I1616_LEQ (NTO_I1616 a, NTO_I1616 b) {
  return (a <= b);
}

//-----------------------------------------------------------------------------------
// Return 1 if a is greater than b, where a and b are NTO_I1616 fixed point values;
// return zero otherwise
//-----------------------------------------------------------------------------------
NTO_INLINE static NTO_I32 I1616_GT (NTO_I1616 a, NTO_I1616 b) {
  return (a > b);
}

//-----------------------------------------------------------------------------------
// Return 1 if a is greater than or equal to b, where a and b are NTO_I1616 fixed
// point values; return zero otherwise
//-----------------------------------------------------------------------------------
NTO_INLINE static NTO_I32 I1616_GEQ (NTO_I1616 a, NTO_I1616 b) {
  return (a >= b);
}

//-----------------------------------------------------------------------------------
// Return 1 if a is equal to b, where a and b are NTO_I1616 fixed point values;
// return zero otherwise
NGTHSTATIC NTO_I32 I1616_EQ (NTO_I1616 a, NTO_I1616 b) { return (a == b); }

// Return 1 if a is not equal to b, where a and b are NTO_I1616 fixed point values;
// return zero otherwise
NGTHSTATIC NTO_I32 I1616_NEQ (NTO_I1616 a, NTO_I1616 b) { return (a != b); }

// Compute and return the absolute value of x (i.e., abs(x)). Both the input x and
// the computed result are NTO_I1616 fixed point values.
// Note that I1616_ABS() does not compute the correct answer when x has the
// mathematical value -32768 (hexadecimal value 0x80000000). The correct answer is
// 32768, which overflows the NTO_I1616 fixed point representation. In this case,
// I1616_ABS() returns -32768 (i.e., the same value as the input).
NGTHSTATIC NTO_I1616 I1616_ABS (NTO_I1616 x) { return ((x < 0) ? -x : x); }

// Compute and return the negative absolute of x (i.e., -abs(x)). Both the input x
// and the computed result are NTO_I1616 fixed point values.
NGTHSTATIC NTO_I1616 I1616_NEGABS (NTO_I1616 x) { return ((x < 0) ? x : -x); }

// Compute and return -x. Both the input x and the computed result are NTO_I1616
// fixed point values.
NGTHSTATIC NTO_I1616 I1616_NEGATE (NTO_I1616 x) { return (-x); }

// Compute and return floor(x). Both the input x and the computed result are
// NTO_I1616 fixed point values.
NGTHSTATIC NTO_I1616 I1616_FLOOR (NTO_I1616 x) { return (x & 0xffff0000); }

// Convert an NTO_I1616 fixed point value f to a 32-bit signed integer and return
// the result. If f has a non-zero fractional component, the value is rounded
C:\Users\perry\Desktop\MiniNitro\Nitro\FixedMath.h

// towards negative infinity.
// All assembly and C implementations produce bit-identical results.
// Performance notes:
// Intel Centrino Core Duo T2500 (2 MB L2, 2.0 GHz, FSB 677 MHz), MSVC 6 compiler,
// Release mode:
// - NTO_MATH_FIXED_C_64 is 1.0x as fast as NTO_MATH_FIXED_C_32
// - NTO_MATH_FIXED_ASM_X86 is 2.7x as fast as NTO_MATH_FIXED_C_64
// - NTO_MATH_FIXED_ASM_X86 is 2.7x as fast as NTO_MATH_FIXED_C_32
//------------------------------------------------------------------------------
NTO_INLINE static NTO_I32 I1616_TO_I32 (NTO_I1616 f)
{
    #if (NTO_MATH_MODE == NTO_MATH_FIXED_C_64)
    {
        //----bits, which has the effect of rounding f towards negative infinity.
        return((NTO_I32) (f >> 16));
    }
    #elif (NTO_MATH_MODE == NTO_MATH_FIXED_C_32)
    {
        //----C implementation (32-bit and 64-bit). Perform an arithmetic right shift
        //----of 16 bits, which has the effect of rounding f towards negative
        //----infinity. Note that ANSI C does not require that sign extension be
        //----performed when the left operand of a right shift is a signed integer.
        //----Therefore, sign extension is performed manually to ensure a portable
        //----implementation.
        NTO_I32 sign = (f & 0x80000000);
        f >>= 16;
        if (sign) f |= 0xffff0000;
        return(f);
    }
    #elif (NTO_MATH_MODE == NTO_MATH_FIXED_ASM_X86)
    {
        //----x86 assembly implementation
        _asm

            //----Set eax to the parameter f
            mov eax, f;

            //----Arithmetic right shift by 16 bits, which has the effect of rounding
            //----f towards negative infinity
            sar eax, 16;

    }
    #endif
}

//------------------------------------------------------------------------------
// Convert a 32-bit signed integer i to an NTO_I1616 fixed point value and return
// the result. If i underflows or overflows the NTO_I1616 representation, the result
// is undefined.
//------------------------------------------------------------------------------
NTO_INLINE static NTO_I1616 I32_TO_I1616 (NTO_I32 i)
{
    return(i << 16);
}

//------------------------------------------------------------------------------
// Compute and return the signed sum (a + b). The inputs a and b and the computed
// sum are NTO_I1616 fixed point values. If the sum overflows the NTO_I1616
// representation, the result is undefined.
NTO_INLINE static NTO_I1616 I1616_ADD (NTO_I1616 a, NTO_I1616 b) {
    return(a + b);
}

// Compute and return the signed difference (a - b). The inputs a and b and the
// computed difference are NTO_I1616 fixed point values. If the difference overflows
// the NTO_I1616 representation, the result is undefined.
NTO_INLINE static NTO_I1616 I1616_SUB (NTO_I1616 a, NTO_I1616 b) {
    return(a - b);
}

// Compute and return the signed product of a and b (i.e., a * b). The inputs a and
// b are NTO_I1616 fixed point values, and the computed result is an NTO_I2408 fixed
// point value. The signed product is rounded towards negative infinity.
// All assembly and C implementations produce bit-identical results.
// Performance notes:
// Intel Centrino Core Duo T2500 (2 MB L2, 2.0 GHz, FSB 677 MHz), MSVC 6 compiler,
// Release mode:
// - NTO_MATH_FIXED_C_64 is ~2.6x as fast as NTO_MATH_FIXED_C_32
// - NTO_MATH_FIXED_ASM_X86 is ~1.6x as fast as NTO_MATH_FIXED_C_64
// - NTO_MATH_FIXED_ASM_X86 is ~4.2x as fast as NTO_MATH_FIXED_C_32
NTO_INLINE static NTO_I2408 I1616_MUL_I2408 (NTO_I1616 a, NTO_I1616 b) {
    #if (NTO_MATH_MODE == NTO_MATH_FIXED_C_64)

        //-----C implementation (64-bit). Compute the 64-bit product and normalize the
        //-----result to the NTO_I2408 fixed point format. The low 24 fractional bits
        //-----of the 64-bit product are discarded, effectively rounding the value
        //-----towards negative infinity.
        return(((NTO_I32) (((NTO_I64) a) * ((NTO_I64) b) >> 24)));
    #elif (NTO_MATH_MODE == NTO_MATH_FIXED_C_32)
        //-----C implementation (32-bit)
        NTO_U32 z, zHigh, zLow;
        NTO_I32 result;
        NTO_I32 aSign = (a & 0x80000000);
        NTO_I32 bSign = (b & 0x80000000);
        a = (aSign) ? -a : a;
        b = (bSign) ? -b : b;
        UMUL64(*((NTO_U32*) &a, *(NTO_U32*) &b, &zHigh, &zLow);
        //-----Merge the integer and fractional bits of the unsigned product
        z = (zLow >> 24) | (zHigh << 8);
    }
Compute the signed product
result = ((aSign ^ bSign) ? -(NTO_I32 z) : z);

If the signed product is negative and the low 24 bits of the 64-bit product are not all zero, then subtract 1 from the result to round the result towards negative infinity. To see why it is necessary to subtract 1, consider rounding the value -1.5 towards negative infinity. Simply dropping the fractional portion produces the value -1.0, which has the effect of rounding -1.5 towards zero. However, dropping the fractional portion and then subtracting 1 (i.e., -1.0 - 1 = -2.0) produces the desired result of rounding -1.5 towards negative infinity.

if ((aSign ^ bSign) && (zLow & 0x00ffffff)) result -= 1;

Return the signed product
return(result);

//---x86 assembly implementation
__asm { 
      //----Set eax and edi to the parameters a and b, respectively
mov   eax, a;
mov   edi, b;

      //----Set [edx:eax] to the 64-bit product of a and b
imul  edi;

      //----Shift the product's 24 integer bits into bits 31:8 of edx
shl   edx, 8;

      //----Shift the product's 8 fractional bits into bits 7:0 of eax. The low 24 fractional bits of the 64-bit product are discarded, effectively rounding the value towards negative infinity.
shr   eax, 24;

      //----Merge and return the integer and fractional bits
or    eax, edx;
} 
} #endif

//---Compute and return the signed product of a and b (i.e., a * b). The inputs a and b and the computed product (a * b) are NTO_I1616 fixed point values. The signed product is rounded towards negative infinity.
//
// All assembly and C implementations produce bit-identical results.
//
// Performance notes:
//
// Intel Centrino Core Duo T2500 (2 MB L2, 2.0 GHz, FSB 677 MHz), MSVC 6 compiler,
// Release mode:
//   - NTO_MATH_FIXED_C_64 is ~2.7x as fast as NTO_MATH_FIXED_C_32
//   - NTO_MATH_FIXED_ASM_X86 is ~1.5x as fast as NTO_MATH_FIXED_C_64
//   - NTO_MATH_FIXED_ASM_X86 is ~4.1x as fast as NTO_MATH_FIXED_C_32

//-----------------------------------------------------------------------------------
NTO_INLINE static NTO_I1616 I1616_MUL (NTO_I1616 a, NTO_I1616 b)
{
    //----Compute the signed product of a and b
    #if (NTO_MATH_MODE == NTO_MATH_FIXED_C_64)
    
    //-----C implementation (64-bit). Compute the 64-bit product and normalize the
    //-----result to the NTO_I1616 fixed point format. The low 16 fractional bits
    //-----of the 64-bit product are discarded, effectively rounding the value
    //-----towards negative infinity.
    return(((NTO_I32) (((NTO_I64) a) * ((NTO_I64) b)) >> 16));
    
    #elif (NTO_MATH_MODE == NTO_MATH_FIXED_C_32)
    
    //-----C implementation (32-bit)
    NTO_U32 z, zHigh, zLow;
    NTO_I32 result;

    //-----Determine the signs of parameters a and b
    NTO_I32 aSign = (a & 0x80000000);  // high bit is 1 if a is negative
    NTO_I32 bSign = (b & 0x80000000);  // high bit is 1 if b is negative

    //-----Make parameters a and b unsigned
    a = (aSign) ? -a : a;
    b = (bSign) ? -b : b;

    //-----Compute the 64-bit unsigned product
    UMUL64(*((NTO_U32*) &a, *((NTO_U32*) &b, &zHigh, &zLow));

    //-----Merge the integer and fractional bits of the unsigned product
    z = (zLow >> 16) | (zHigh << 16);

    //-----Compute the signed result
    result = ((aSign ^ bSign) ? -((NTO_I32) z) : z);

    //-----If the signed product is negative and the low 16 bits of the 64-bit
    //-----product are not all zero, then subtract 1 from the result to round the
    //-----result towards negative infinity. To see why it is necessary to
    //-----subtract 1, consider rounding the value -1.5 towards negative infinity.
    //-----Simply dropping the fractional portion produces the value -1.0, which
    //-----has the effect of rounding -1.5 towards zero. However, dropping the
    //-----fractional portion and then subtracting 1 (i.e., -1.0 - 1 = -2.0)
    //-----produces the desired result of rounding -1.5 towards negative infinity.
    if ((aSign ^ bSign) & (zLow & 0x0000ffff)) result -= 1;

    //-----Return the signed product
    return(result);
    
    #elif (NTO_MATH_MODE == NTO_MATH_FIXED_ASM_X86)
    
    //-----x86 assembly implementation
    __asm {

        //-----Set eax and edi to the parameters a and b, respectively
        mov eax, a;
        mov edi, b;

        //-----Set [edx:eax] to the 64-bit product of a and b
        imul edi;
    }
Compute and return the positive square root of \( f \). Where both \( f \) and the computed result are non-negative NTO_I1616 fixed point values.

Special cases are handled as follows. If \( f \) is zero, \( \text{I1616\_SQR}() \) returns exactly zero. If \( f \) is negative, the result is undefined.

All assembly and C implementations produce bit-identical results.

NTO_I1616 I1616_SQR (NTO_I1616 \( f \));

Compute and return the positive square root of \( f \), where \( f \) is a non-negative NTO_I2408 fixed point value and the computed result is an NTO_I1616 fixed point value.

NTO_I2408 I1616_SQR (NTO_I1616 \( f \));
// value.
//
// Special cases are handled as follows. If f is zero, I2408_SQRT_I1616() returns
// exactly zero. If f is negative, the result is undefined.
//
// All assembly and C implementations produce bit-identical results.
//
NTO_I1616 I2408_SQRT_I1616 (NTO_I2408 f);

// Compute and return the signed quotient (n / d). The input numerator n, the input
// denominator d, and the computed quotient are NTO_I1616 fixed point values. The
// quotient is rounded towards zero.
//
// On output, I1616_DIV() sets status to NTO_FIXED_MATH_NO_ERROR,
// NTO_FIXED_MATH_OVERFLOW, NTO_FIXED_MATH_UNDERFLOW, or NTO_FIXED_MATH_NAN,
// depending on the outcome of the quotient computation. The possible cases are as
// follows:
//
// 1. If n is any value and d is zero, I1616_DIV() returns zero and sets status to
//    NTO_FIXED_MATH_NAN.
//
// 2. If n is zero and d is non-zero, I1616_DIV() returns zero and sets status to
//    NTO_FIXED_MATH_NO_ERROR.
//
// 3. If n is non-zero and d is 0x10000 (i.e., the mathematical value 1),
//    I1616_DIV() returns n and sets status to NTO_FIXED_MATH_NO_ERROR.
//
// 4. If n is non-zero and d is 0xffffffff (i.e., the mathematical value -1),
//    I1616_DIV() returns -n and sets status to NTO_FIXED_MATH_NO_ERROR.
//
// 5. If n and d are both non-zero and the quotient (n / d) overflows the
//    NTO_I1616 fixed point representation, I1616_DIV() returns zero and sets
//    status to NTO_FIXED_MATH_OVERFLOW.
//
// 6. If n and d are both non-zero and the quotient (n / d) underflows the
//    NTO_I1616 fixed point representation, I1616_DIV() returns zero and sets
//    status to NTO_FIXED_MATH_UNDERFLOW.
//
// 7. In all other cases, I1616_DIV() computes and returns the signed fixed point
//    quotient (n / d) and sets status to NTO_FIXED_MATH_NO_ERROR.
//
// All assembly and C implementations produce bit-identical results.
//
// Performance notes:
//
// Intel Centrino Core Duo T2500 (2 MB L2, 2.0 GHz, FSB 677 MHz): MSVC 6 compiler,
// Release mode:
//  - NTO_MATH_FIXED_C_64 is ~1.2x as fast as NTO_MATH_FIXED_C_32
//  - NTO_MATH_FIXED_ASM_X86 is ~1.4x as fast as NTO_MATH_FIXED_C_64
//  - NTO_MATH_FIXED_ASM_X86 is ~1.7x as fast as NTO_MATH_FIXED_C_32
//
// 
#define NTO_FIXED_MATH_NO_ERROR    0
#define NTO_FIXED_MATH_OVERFLOW     1
#define NTO_FIXED_MATH_UNDERFLOW    2
#define NTO_FIXED_MATH_NAN          3

NTO_I1616 I1616_DIV (NTO_I1616 n, NTO_I1616 d, NTO_I32 *status);

// End of C++ wrapper

#ifdef __cplusplus

Newton-Raphson iteration is a method for solving equations numerically. Given a good initial approximation of the solution to an equation, Newton-Raphson iteration converges rapidly on that solution. Convergence is usually quadratic with the number of valid bits in the result roughly doubling with each iteration.

Newton-Raphson iteration method applies to any equation of the form \( f(x) = 0 \), where \( f(x) \) is a differentiable function with derivative \( f'(x) \). The method begins with an approximation \( x(i) \) to a solution \( x \) of the equation. Then the following iterative equation is applied to obtain a better approximation \( x(i+1) \):

\[
x(i+1) = x(i) - \frac{f(x(i))}{f'(x(i))}
\]

For example, consider using the Newton-Raphson iteration method to solve the equation \( f(x) = 0.64 - x^2 = 0 \). The derivative of \( f(x) \) is \( f'(x) = -2x \). Therefore, the Newton-Raphson iteration equation to solve \( f(x) \) is:

\[
x(i+1) = x(i) - \frac{(0.64 - x(i)^2) / (-2 * x(i))}{(0.32 / x(i)) + (0.5 * x(i))}
\]

Let \( x(0) = 1 \) be the initial approximation to the solution of \( f(x) = 0 \). Applying the above formula once produces a better approximation \( x(1) \):

\[
x(1) = (0.32 / x(0)) + (0.5 * x(0)) = (0.32 / 1) + (0.5 * 1) = 0.82
\]

Applying the above formula again produces an even better approximation \( x(2) \):
Applying the above formula again produces an even better approximation $x$.

The Newton-Raphson iteration method is used by this fixed point math implementation to compute reciprocal square roots. The reciprocal square root of $n$ is $\text{rsqrt}(n) = 1 / \sqrt{n}$, which is equivalent to $n^{-0.5}$. Computing the reciprocal square root can be accomplished by solving the equation $f(x) = n - x^2 = 0$, which has a positive solution $x = n^{0.5}$. The derivative of $f(x)$ is $f'(x) = 2 \cdot x^{-3}$. Therefore, the Newton-Raphson iteration equation to solve $f(x) = 0$ is:

$$x_{i+1} = x_i - \left(\frac{n - x_i^2}{2 \cdot x_i^3}\right)$$

For example, consider using the Newton-Raphson iteration method to compute the reciprocal square root of 9. The Newton-Raphson iteration equation is:

$$x_{i+1} = 0.5 \cdot x_i \cdot (3 - n \cdot x_i^2)$$

Let $x[0] = 0.3$ be the initial approximation to the solution of $f(x) = 0$. Applying the above formula once produces a better approximation $x[1]$:

$$x[1] = 1.5 \cdot x[0] \cdot (1 - 3 \cdot x[0]^2)$$

Applying the above formula again produces an even better approximation $x[2]$:

$$x[2] = 1.5 \cdot x[1] \cdot (1 - 3 \cdot x[1]^2)$$

With repeated applications of the above formula, the Newton-Raphson iteration converges to the true solution: $1/3$.

The Newton-Raphson iteration method is also used by this fixed point math implementation to compute reciprocals. The reciprocal of $n$ is $1 / n$, which is equivalent to $n^{-1}$. Computing the reciprocal can be accomplished by solving the equation $f(x) = n - x = 0$, which has a solution $x = n^{-1}$. The derivative of $f(x)$ is $f'(x) = x^{-2}$. Therefore, the Newton-Raphson iteration equation to solve $f(x) = 0$ is:

$$x_{i+1} = x_i - \left(\frac{n - x_i}{2 \cdot x_i^2}\right)$$

For example, consider using the Newton-Raphson iteration method to compute the reciprocal of 0.8. The Newton-Raphson iteration equation is:

$$x_{i+1} = x_i \cdot (2 - n \cdot x_i)$$

Let $x[0] = 1$ be the initial approximation to the solution of $f(x) = 0$. Applying the above formula once produces a better approximation $x[1]$

$$x[1] = x[0] \cdot (2 - 0.8 \cdot x[0])$$

Applying the above formula again produces an even better approximation $x[2]$: 

```
The table contains Performance notes
Implementation notes

Performance notes:
Intel Centrino Core Duo T2500 (2 MB L2, 2.0 GHz, FSB 677 MHz): MSVC 6 compiler,
Release mode:
- NTO_MATH_FIXED_C_64 is ~1.7x as fast as NTO_MATH_FIXED_C_32
- NTO_MATH_FIXED_ASM_X86 is ~1.4x as fast as NTO_MATH_FIXED_C_64
- NTO_MATH_FIXED_ASM_X86 is ~2.4x as fast as NTO_MATH_FIXED_C_32

rsqTable[] is the reciprocal square root lookup table used by the RSQ() function.
The table contains 96 elements and is precomputed as follows:
The purpose of rsqTable[] is to provide a fast and reasonably accurate initial estimate to the reciprocal square root of a number n.

Assume that n is a 0.32 fixed point value that lies in the mathematical range (0.25, 1). Considered as an unsigned integer, n is greater than 2^30. Therefore, either bit 31 or bit 30 of n is 1.

The reciprocal square root of each value in the range (0.25, 1) can be expressed as a 24.8 fixed point value, where the integer portion of the 24.8 fixed point value is always 1. For example, the reciprocal square root of 0.5 can be expressed as the 24.8 fixed point value 0x16a, which has the mathematical value 1.4140625. Consequently, the table only needs to store the low 8 bits (i.e., the fractional bits) of the reciprocal square root of each value in the range (0.25, 1). Storing the integer portion of the 24.8 fixed point value in the table is unnecessary because it is always 1.

For a good tradeoff between table size and the accuracy of the initial estimate, this implementation uses the leading seven fractional bits of n to index into the table, thereby limiting the table size to 128 elements.

Observe that the index into the table is the 7-bit unsigned integer obtained from the leading seven fractional bits of n. Since either bit 6 (i.e., the MSB) or bit 5 of the 7-bit unsigned integer index is 1, the value of the index must be at least 32. Therefore, the index lies in the range [32, 127]. As a result, the table only needs to contain 127 - 32 + 1 = 96 elements and can be indexed using integers that lie in the range [0, 95].

Computing the table is accomplished with the following steps:

First, map an integer index i in the range [0, 95] to the range [0.25234375, 0.99453125], which is an approximation to the range (0.25, 1):

\[(i + 32.3) / 128.0\]

The value of 32.3 (instead of 32.0) is chosen for technical reasons explained below. Next, compute the reciprocal square root:

\[1 / \sqrt{(i + 32.3) / 128.0}\]

Scale the result by 256.0 and round to the nearest integer to obtain a 24.8 fixed point value:

\[\text{round}(256.0 \times \sqrt{(i + 32.3) / 128.0})\]

Finally, subtract (i.e., remove) the integer portion of the 24.8 fixed point value because the integer portion is always 1. Putting all the steps together yields the following formula:

\[\text{rsqTable}[i] = \text{round}(256.0 / \sqrt{(i + 32.3) / 128.0}) - 256\]

The reason for choosing 32.3 instead of 32.0 is to handle the case where the index i is zero. Suppose the value of 32.0 is used instead. Then

\[\text{rsqTable}[0] = \text{round}(256.0 / \sqrt{(0 + 32.0) / 128.0}) - 256\]
\[= \text{round}(256.0 / \sqrt{0.25}) - 256\]
\[= \text{round}(256.0 / 0.5) - 256\]
\[= 512 - 256\]
\[= 256\]

The value 256 requires 9 bits to store. By choosing a value slightly larger than 32.0, such as 32.3, the following result is obtained:
The value 254 requires only 8 bits to store. Therefore the entire table can be stored in just 96 bytes. The error in the initial estimate introduced by choosing 32.3 instead of 32.0 is corrected by the Newton-Raphson iterations following the table lookup.

Although the above discussion has assumed that n lies in the mathematical range (0.25, 1), the RSQ() function uses rsqTable[] to compute reciprocal square roots of values that lie in the mathematical range [0.25, 1] (note the inclusion of 0.25). The first element of rsqTable[] (i.e., rsqTable[0]) is an estimate of the reciprocal square root of 0.25.

static const NTO_U8 rsqTable[] = {
    0xfe, 0xf6, 0xe7, 0xda, 0xd4, 0xce, 0xc9, 0xc3, 0xb8, 0xb3, 0xae, 0xaa, 0xa5, 0x9c, 0x94, 0x90, 0x8d, 0x89, 0x85, 0x82, 0x7f, 0x7b, 0x75, 0x72, 0x6f, 0x6c, 0x66, 0x64, 0x61, 0x5e, 0x5c, 0x59, 0x57, 0x55, 0x52, 0x4e, 0x4c, 0x49, 0x47, 0x45, 0x43, 0x3f, 0x3d, 0x3b, 0x3a, 0x38, 0x36, 0x34, 0x32, 0x31, 0x2f, 0x2d, 0x2c, 0x2a, 0x29, 0x27, 0x24, 0x23, 0x21, 0x20, 0x1e, 0x1d, 0x1c, 0xa9, 0x18, 0x16, 0x15, 0x14, 0x13, 0x11, 0x10, 0x0f, 0x0e, 0xd0, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
};

static NTO_U32 RSQ (NTO_U32 n)
{
    if (NTO_MATH_MODE == NTO_MATH_FIXED_C_32) ||
        (NTO_MATH_MODE == NTO_MATH_FIXED_C_64)
    {
        //-----C implementation (32-bit and 64-bit). x0 is the initial estimate
        //-----obtained by a table lookup. x1 is the estimate obtained after the first
        //-----Newton-Raphson iteration. x2 is the estimate obtained after the second
        //-----Newton-Raphson iteration.
        NTO_U32 x0, x1, x2;

        //-----Perform a table lookup on the leading 7 bits of n to obtain the 8
        //-----fractional bits of a 24.8 fixed point estimate to rsq(n) and add the
        //-----integer portion of the 24.8 fixed point estimate (i.e., 0x100), which
        //-----always has the mathematical value 1. The result is a 24.8 fixed point
        //-----estimate to rsq(n). Running example: consider the case where n is
        //-----0x80000000. Considered as a 0.32 fixed point value, n has the
        //-----mathematical value (0x80000000 / 2^32) = 0.5. x0 = rsqTable[(n >> 25) -
        //-----32] + 0x100 = rsqTable[(0x80000000 >> 25) - 32] + 0x100 = rsqTable[32]
        //-----+ 0x100 = 0x69 + 0x100 = 0x169, which has the mathematical value (0x169
        //-----/ 2^8) = 1.41015625. This is the initial estimate of the reciprocal
        //-----square root of 0.5.
        x0 = rsqTable[(n >> 25) - 32] + 0x100;

        //-----Begin the first Newton-Raphson iteration. Compute the 16.16 fixed point
        //-----value (x0 * x0). Running example: x1 = x0 * x0 = 0x169 * 0x169 =
        //-----0x1fd11, which has the mathematical value (0x1fd11 / 2^16) =
        //-----1.9885406494140625.
        x1 = x0 * x0;

        //-----Convert x0 from a 24.8 fixed point value to a 17.15 fixed point value.
    }
//------Running example: x0 = (x0 << 7) = (0x169 << 7) = 0xb480, which has the
//------mathematical value (0xb480 / 2^8) = 1.41015625.
x0 <<= 7;

//------Convert n from a 0.32 fixed point value to a 17.15 fixed point value
//--------and compute the 17.15 fixed point value (n * x0 * x0). Running example:
//-------n >> 17 = 0x80000000 >> 17 = 0x4000. (n >> 17) * x1 = 0x4000 * 0xfd11
//------0x0000000007f444000. Extracting the middle 32 bits of the 64-bit
//------product yields x1 = 0x00007f44, which has the mathematical value
//------(0x7f44 / 2^15) = 0.9942626953125.
x1 = UMUL64_MID32(n >> 17, x1);

//------Compute the 17.15 fixed point value (3 - (n * x0 * x0)). Running
//------example: x1 = (3 << 15) - x1 = 0x18000 - 0x7f44 = 0x100bc, which has
//------the mathematical value (0x100bc / 2^15) = 2.0057373046875.
x1 = (3 << 15) - x1;

//------Compute the 1.31 fixed point value (0.5 * x0 * (3 - (n * x0 * x0))).
//------Note that the multiplication below of two 17.15 fixed point values
//------produces a 64-bit 34.30 fixed point value. Only the low 32 bits of the
//------64-bit product are kept, thereby truncating the 34.30 fixed point value
//------to a 2.30 fixed point value. The multiplication by 0.5 effectively
//------shifts the binary point to the left by one bit, which makes x1 a 1.31
//------fixed point value. This is the end of the first Newton-Raphson
//------iteration. Running example: x1 = x0 * x1 = 0xb480 * 0x100bc =
//------0xb5048e00, which has the mathematical value (0xb5048e00 / 2^310 =
//------1.41201498316162109375. Note that x1 is a better estimate of the
//------reciprocal square root of 0.5 than x0.
x1 = x0 * x1;

//------Begin the second Newton-Raphson iteration. Compute the 1.31 fixed point
//------value (n * x1). Running example: x2 = (n * x1) = (0x80000000 *
//------0xb5048e00) = 0x5a82470000000000. Keeping only the high 32 bits of the
//------64-bit product yields the result x2 = 0x5a824700, which has the
//------mathematical value (0x5a824700 / 2^31) = 0.70710074901580810546875.
x2 = UMUL64_HIGH32(n, x1);

//------Compute the 2.30 fixed point value (n * x1 * x1). Running example: x2 =
//------(x2 * x1) = (0x5a824700 * 0xb5048e00) = 0x3fff8705f6f0000. Keeping
//------only the high 32 bits of the 64-bit product yields the result x2 =
//------0x3fff8705, which has the mathematical value (0x3fff870 / 2^30) =
//------0.9998829381704330443359375.
x2 = UMUL64_HIGH32(x2, x1);

//------Compute the 2.30 fixed point value (3 - (n * x1 * x1)). Running
//------example: x2 = (3 << 30) - x2 = (0xc0000000) - (0x3fff870) =
//------0x80004790, which has the mathematical value (0x80004790 / 2^30) =
//------2.00001706182956695556640625.
x2 = (3 << 30) - x2;

//------Compute the 2.30 fixed point value (0.5 * x1 * (3 - (d * x1 * x1))).
//------Note that the multiplication below of a 1.31 fixed point value and a
//------2.30 fixed point value produces a 64-bit 3.61 fixed point value. Only
//------the high 32 bits of the 64-bit product are kept, thereby truncating the
//------3.61 fixed point value to a 3.29 fixed point value. The multiplication
//------by 0.5 effectively shifts the binary point to the left by one bit,
//------which makes x2 a 2.30 fixed point value. This is the end of the second
//------Newton-Raphson iteration. Running example: x2 = (x1 * x2) = (0xb5048e00
//------* 0x80004790) = 0x5a82799a155f1e00. Keeping only the high 32 bits of
//------the 64-bit product yields the result 0x5a82799a, which has the
//----mathematical value  (0x5a82799a / 2^30) =
//----1.41421356238424777984619140625. The true reciprocal square root of 0.5
//----is approximately 1.4142135623730950488016887242097. In this example,
//----the computed result x2 is accurate to within 0.000000000012 of the true
//----answer.
x2 = UMUL64_HIGH32(x1, x2);

//----Return the result
return(x2);
#
elif (NTO_MATH_MODE == NTO_MATH_FIXED_ASM_X86) {

  //----x86 assembly implementation
  __asm {

    //----Set ebx to the input n
    mov  ebx, n;

    //----Perform a table lookup on the leading 7 bits of n to obtain the 8
    //----fractional bits of a 24.8 fixed point estimate to rsq(n) and add
    //----the integer portion of the 24.8 fixed point estimate (i.e., 0x100),
    //----which always has the mathematical value 1. The result is a 24.8
    //----fixed point estimate to rsq(n).
    mov  edi, ebx;
    shr  edi, 25;
    sub  edi, 32;
    mov  eax, 0;
    mov  al, rsqTable[edi];
    add  eax, 0x100;

    //----Let x0 be the current value of eax, which is a 24.8 fixed point
    //----estimate to rsq(n). Copy eax to esi.
    mov  esi, eax;

    //----Begin the first Newton-Raphson iteration. Compute the 16.16 fixed
    //----point value [edx:eax] = (x0 * x0).
    imul  eax;

    //----Convert n from a 0.32 fixed point value to a 17.15 fixed point
    //----value and compute the 17.15 fixed point value (n * x0 * x0).
    mov  edi, ebx;
    shr  edi, 17;
    imul  edi;
    shr  eax, 16;
    shl  edx, 16;
    or   eax, edx;

    //----Convert x0 (i.e., esi) from a 24.8 fixed point value to a 17.15
    //----fixed point value
    shl  esi, 7;

    //----Compute the 17.15 fixed point value (3 - (n * x0 * x0))
    sub  eax, 0x18000;
    neg  eax;

    //----Compute the 1.31 fixed point value (0.5 * x0 * (3 - (n * x0 *
    //----x0))). Note that the multiplication below of two 17.15 fixed point
    //----values produces a 64-bit 34.30 fixed point value. Only the low 32
square roots of general fixed point values

rsqrt

for the normalization of $M$ in step such that $R$ is a specialized function that only handles a restricted set of fixed point

mul esi, eax;

---

Compute the 2.30 fixed point value $(n \times x_l)$
mul eax, edx;

---

Compute the 2.30 fixed point value $(3 - (n \times x_l))$
mov eax, edx;
sub eax, 0xc0000000;
neg eax;

---

Compute the 2.30 fixed point value $(0.5 \times x_l \times (3 - (d \times x_l))$)

---

The `RSQ()` function (see above) computes the reciprocal square root of a 0.32
fixed point value that lies in the mathematical range $(0.25, 1)$. In other words,
`RSQ()` is a specialized function that only handles a restricted set of fixed point
input values. This section describes how to use `RSQ()` to compute reciprocal
square roots of general fixed point values.

First, consider the following algorithm for computing the reciprocal square root
of any (real-valued) number $M$:

1. Normalize $M$ to the range $(0.25, 1)$ by choosing a normalization exponent $e$
such that $(M / 2^e)$ lies in the range $(0.25, 1)$.

2. Compute the reciprocal square root $R$ of the normalized value of $M$ (i.e.,
compute $R = \text{rsqrt}(M / 2^e) = 1 / \sqrt{M / (2^e)} = \sqrt{2^e / M}$).

3. Compute the reciprocal square root of $M$ by unnormalizing $R$ (i.e., compensate
for the normalization of $M$ in step 1). To unnormalize $R$, note that $R =
\sqrt{2^e / M} = \sqrt{2^e} / \sqrt{M} = 2^{(e/2)} \times \text{rsqrt}(M)$. It follows that
$\text{rsqrt}(M) = R \times 2^{(-e/2)}$. Therefore, computing the reciprocal square root of
M simply requires multiplying R by \(2^{(-e/2)}\).

This algorithm can be translated to a fixed point implementation as follows.

Suppose that the value M in the above algorithm is actually the mathematical value of a 32-bit I.F fixed point value N (i.e., N has I integer bits and F fractional bits, where I + F = 32 and M = (N / (2^F))). The following three steps compute the reciprocal square root of N:

a. Normalize N by shifting N to the left by s bits, where s is an integer chosen so that \((N \cdot 2^s)\) is at least 2^30. Note that this is equivalent to choosing s such that \((N \cdot 2^s)\) is a 0.32 fixed point value that lies in the mathematical range \([0.25, 1)\) (i.e., \(0.25 < (N \cdot 2^s) / 2^{32} < 1\)). The relationship between s and the exponent e (see step 1 above) will be explained in step c below.

b. Compute the reciprocal square root of the normalized value of N by calling the RSQ() function with \((N \cdot 2^s)\) as its argument. The result R is a 2.30 fixed point value that represents the reciprocal square root of the normalized value of N.

c. Unnormalize R by shifting R to the right by \((16 - 0.5 \cdot (F + s))\) bits. To derive this formula, recall from step 3 that unnormalizing R requires multiplying R by \(2^{(-e/2)}\), where e is the normalization exponent chosen so that \((M / (2^e))\) lies in the range \([0.25, 1)\). Note that the normalized value \((M / (2^e))\) is equal to the normalized value \((N \cdot 2^s / 2^{32})\) (see step a above):

\[
M / 2^e = N \cdot 2^s / 2^{32}
\]

\[
= N \cdot 2^{(s-32)}
\]

\[
= M \cdot 2^F \cdot 2^{(s-32)}
\]

\[
= M \cdot 2^{(F+s-32)}
\]

\[
= M \cdot 2^{(F+s-32)}
\]

Note that the third equation above uses the fact that \(N = M \cdot 2^F\). By equating the exponents, it follows that \(e = 32 - F - s\), where F is the number of fractional bits of the fixed point value N, and s is the integer chosen above in step a. Multiplication by \(2^{(-e/2)}\) can be implemented as a right shift by \(e/2 = (16 - 0.5 \cdot (F + s))\) bits.

Note that the functions I2408_RSQ_I1616() and I1616_RSQ() compute their results as 16.16 fixed point values. Since the RSQ() function computes its result as a 2.30 fixed point value, the conversion to a 16.16 fixed point value requires an additional right shift of 14 bits.

For the I2408_RSQ_I1616() function, the input N is a 24.8 fixed point value (i.e., \(I = 24, F = 8\)). Therefore, the computed 2.30 fixed point reciprocal square root R is unnormalized and converted to a 16.16 fixed point value by shifting R to the right by \((16 - (0.5 \cdot (8 + s))) + 14\) = \((26 - (s / 2))\) bits.

For the I1616_RSQ() function, the input N is a 16.16 fixed point value (i.e., \(I = 16, F = 16\)). Therefore, the computed 2.30 fixed point reciprocal square root R is unnormalized and converted to a 16.16 fixed point value by shifting R to the right by \((16 - (0.5 \cdot (16 + s))) + 14\) = \((22 - (s / 2))\) bits.

Since I2408_RSQ_I1616() and I1616RSQ() must divide s by 2 in step c, both functions choose an even value of s in step a, thereby allowing the division of s by 2 to be implemented as a right shift (i.e., \((s >> 1)\)).
All assembly and C implementations produce bit-identical results, except in the case that the input \( n \) is zero.

Implementation notes:

- \( \text{I2408\_RSQ\_I1616()} \) computes the reciprocal square root of \( n \) using the approach described in the section Fixed Point Implementation Notes On Computing Reciprocal Square Roots (see above).
- The documentation for the C implementation below uses a running example to help explain the fixed point implementation. This running example considers the case that the input value is \( n = 0x4000 \) (i.e., the mathematical value \((0x4000 / 2^8) = 64\)).

Reciprocal Square Roots

```c
NT0_I1616 I2408_RSQ_I1616 (NT0_I2408 n)
{
    NTO_U32 rsq;

    //-----Set b to the input n
    NTO_U32 b = *(NTO_U32 *) &n;

    //-----Determine an even integer s such that the expression \( b << s \) is a 0.32 fixed point value that lies in the mathematical range \([0.25, 1)\). Running example: consider the case where \( b \) is the 24.8 fixed point value 0x4000, which has the mathematical value \((0x4000 / 2^8) = 64\). CountLeadingZeros(0x4000) is 17, because 0x4000 contains 17 zeroes before its leading 1 bit. Therefore, \( s = (17 & 0xffffffff) = 16 \).
    NTO_U32 s = (CountLeadingZeros(b) & 0xffffffff);

    //-----Shift b to the left by \( s \) bits so that \( b \) is a 0.32 fixed point value that lies in the mathematical range \([0.25, 1)\). Running example: \( b = (b << s) = (0x4000 << 16) = 0x4000000000 \), which is a 0.32 fixed point value with the mathematical value \((0x4000000000 / 2^{32}) = 0.25 \).
    b <<= s;

    //-----Compute the reciprocal square root of \( b \). Note that the RSQ() function computes the result as a 2.30 fixed point value. Running example: \( rsq = RSQ(b) = RSQ(0x40000000) = 0x7fffffff \), which has the mathematical value \((0x7fffffff / 2^{30}) = 1.99999999068677425384521484375 \).
    rsq = RSQ(b);

    //-----Unnormalize rsq and convert the result to a 16.16 fixed point value by shifting rsq to the right by \((26 - s/2)\) bits (see the section Fixed Point Implementation Notes On Computing Reciprocal Square Roots for the derivation of this formula). Running example: \( rsq >> (26 - (s >> 1)) = 0x7fffffff >> (26 - (16 >> 1)) = 0x7ffffff >> 18 = 0x1fff \), which is a 16.16 fixed point value with the mathematical value \((0x1fff / 2^{16}) = 0.1249847412109375 \). Note that the true reciprocal square root of 64 is 1/8
    //----= 0.125.
    rsq >>= (26 - (s >> 1));

    //-----Return the result
    return (* (NT0_I1616*) &rsq);
}
```

Compute and return the reciprocal square root of \( f \) (i.e., \( 1 / \sqrt{f} \)), where the input \( n \) and the computed result are NT0_I1616 fixed point values. The computed value is rounded towards negative infinity. If the input \( n \) is less than or equal to zero, the result is undefined.
All assembly and C implementations produce bit-identical results, except in the case that the input n is zero.

Implementation notes:
- I1616 RSQ() computes the reciprocal square root of n using the approach described in the section Fixed Point Implementation Notes On Computing Reciprocal Square Roots (see above).
- The documentation for the C implementation below uses a running example to help explain the fixed point implementation. This running example considers the case where the input value is n = 0x4000 (i.e., the mathematical value (0x4000 / 2^16) = 1/4 = 0.25).

Performance notes:
- Intel Centrino Core Duo T2500 (2 MB L2, 2.0 GHz, FSB 677 MHz), MSVC 6 compiler, Release mode:
  - NTO_MATH_FIXED_C_64 is ~1.7x as fast as NTO_MATH_FIXED_C_32
  - NTO_MATH_FIXED_ASM_X86 is ~1.3x as fast as NTO_MATH_FIXED_C_64
  - NTO_MATH_FIXED_ASM_X86 is ~2.2x as fast as NTO_MATH_FIXED_C_32

---

```c
NTO_I1616_I1616_RSQ (NTO_I1616 n)
{
    NTO_U32 rsq;

    //-----Set b to the input n
    NTO_U32 b = *(NTO_U32 *)&n;

    //-----Determine an even integer s such that the expression (b << s) is a 0.32 fixed point value that lies in the mathematical range [0.25, 1). Running example: consider the case where b is the 16.16 fixed point value 0x4000, which has the mathematical value (0x4000 / 2^16) = 1/4 = 0.25.
    //-----CountLeadingZeros(0x4000) is 17, because 0x4000 contains 17 zeroes before its leading 1 bit. Therefore, s = (17 & 0xffffffff) = 16.
    NTO_U32 s = (CountLeadingZeros(b) & 0xffffffff);

    //-----Shift b to the left by s bits so that b is a 0.32 fixed point value that lies in the mathematical range [0.25, 1). Running example: b = (b << s) = (0x4000 << 16) = 0x400000000, which is a 0.32 fixed point with the mathematical value (0x400000000 / 2^32) = 0.25.
    b <<= s;

    //-----Compute the reciprocal square root of b. Note that the RSQ() function computes the result as a 2.30 fixed point value. Running example: rsq = RSQ(0x7fffffff / 2^30) = 1.9999999906867425384521484375.
    rsq = RSQ(b);

    //-----Unnormalize rsq and convert the result to a 16.16 fixed point value by shifting rsq to the right by (22 - s/2) bits (see the section Fixed Point Implementation Notes On Computing Reciprocal Square Roots for the derivation of this formula). Running example: rsq >> (22 - (s >> 1)) = 0x7ffffff >> (22 - (16 >> 1)) = 0x7ffffff >> 14 = 0x1ffff, which is a 16.16 fixed point value with the mathematical value (0x1ffff / 2^16) = ~1.9999847412109375. Note that the true reciprocal square root of 0.25 is 2.
    rsq >>= (22 - (s >> 1));

    //-----Return the result
    return (*(NTO_I1616*) &rsq);
```
Compute and return the positive square root of n. The input n is a non-negative 32-bit I.F fixed point value (i.e., n has I integer bits and F fractional bits, where I + F = 32) and the computed result is a 32-bit I'.F' fixed point value (i.e., the computed result has I' integer bits and F' fractional bits, where I' + F' = 32).

The values of I, F, I', and F' are not passed directly as inputs to the SQRT() function. Instead, the caller must pass shift as an input to the SQRT() function, where shift is a non-negative integer such that

\[
\text{shift} = 46 - F' + F/2
\]

SQRT() can be called for a wide range of values of I, F, I', and F'. For example, if the input n is a 24.8 fixed point value (i.e., I = 24 and F = 8) and the desired output format is a 16.16 fixed point value (i.e., I' = F' = 16), then the calling function should compute shift as \((46 - 16 + 8/2) = 30 + 8/2 = 30 + 4 = 34\).

If n is zero, SQRT() returns exactly zero.

All assembly and C implementations produce bit-identical results.

Implementation notes:

- SQRT() computes the square root of n as:

\[
\sqrt{n} = n^{1/2}
\]

\[
= n^{1 - 1/2}
\]

\[
= n^1 \times n^{-1/2}
\]

\[
= n \times \sqrt{n}
\]

where rsqrt(n) denotes the reciprocal square root of n.

- SQRT() computes the reciprocal square root of n by first normalizing n to a 0.32 fixed point value that lies in the mathematical range \([0.25, 1]\) and then calling the RSQ() function. The RSQ() function computes the reciprocal square root as a 2.30 fixed point value (see the documentation for RSQ() above for more details).

- To maximize intermediate fractional precision, SQRT() postpones the unnormalization of the computed reciprocal square root until the final step. As explained in the above section Fixed Point Implementation Notes On Computing Reciprocal Square Roots, the unnormalization step requires shifting the computed reciprocal square root to the right by \((16 - 0.5 \times (F + s))\) bits, where F is the number of fractional bits in the input (i.e., n) and s is the normalization shift amount required to normalize n to a 0.32 fixed point value that lies in the mathematical range \([0.25, 1]\).

- Note that n is an I.F fixed point value and that the computed reciprocal square root is a 2.30 fixed point value. Therefore, the product \((n \times \text{rsqrt}(n))\) is a 64-bit \((2+I).(30+F)\) fixed point value. Converting this 64-bit product to a 32-bit I'.F' fixed point value requires shifting the 64-bit product to the right by \((30 + F - F')\) bits.

- Therefore, the unnormalization step and the conversion from a \((2+I).(30+F)\) fixed point value to an I'.F' fixed point value can be combined into a single right shift of \(((30 + F - F') + (16 - 0.5 \times (F + s))\) bits. This expression simplifies to \((46 - F' + F/2 - s/2)\) bits.

- Note that the shift parameter is defined as \((46 - F' + F/2)\). Consequently, the final step of the SQRT() function is to shift the 64-bit product \((n \times \text{rsqrt}(n))\) to the right by \((\text{shift} - s/2)\) bits.
static NTO_U32 SQRT (NTO_U32 n, NTO_I32 shift)
{
    NTO_U32 b;
    NTO_U32 s;
    NTO_U32 rsq;
    NTO_U32 sqrt;
    NTO_U32 sqrtHigh, sqrtLow;

    if (!n) return(0);

    //---Determine an even integer s such that the expression (n << s) is a 0.32
    //---fixed point value that lies in the mathematical range [0.25, 1). Running
    //---example: consider the case where n is the 16.16 fixed point value
    //---0x3e800000, which has the mathematical value (0x3e80000 / 2^16) = 1000.
    //---CountLeadingZeros(0x3e80000) is 6, because 0x3e80000 contains 6 zeroes
    //---before its leading 1 bit. Therefore, s = (6 & 0xffffffff) = 6.
    //---Therefore
    s = (CountLeadingZeros(n) & 0xffffffff);

    //---Normalize n by shifting n to the left by s bits. The computed value b is a
    //---0.32 fixed point value that lies in the mathematical range [0.25, 1). Running
    //---example: b = (n << s) = (0x3e80000 << 6) = 0xfa000000, which is a
    //---0.32 fixed point with the mathematical value (0xf000000 / 2^32) =
    //---0.9765625.
    b = (n << s);

    //---Compute the reciprocal square root of b. Note that the RSQ() function
    //---computes the result as a 2.30 fixed point value. Running example: rsq =
    //---RSQ(b) = RSQ(0xf000000) = 0x40c3713a, which has the mathematical value
    //---(0x40c3713a / 2^30) = 1.01192885078489780426025390625.
    rsq = RSQ(b);

    //---Compute the 64-bit (2+I).(30+F) square root of n as sqrt(n) = n / sqrt(n) =
    //---n * rsqrt(n). Note that n is an I.F fixed point value and that rsq is a
    //---2.30 fixed point value. Therefore, their product is a 64-bit (2+I).(30+F)
    //---fixed point value. Set sqrtHigh and sqrtLow to the high 32 bits and the low
    //---32 bits of the 64-bit product, respectively. Running example: n * rsq =
    //---(0x3e80000 * 0x40c3713a) = 0x0c7f24a90000. Since n is an 16.16 fixed
    //---point value, the product is a 64-bit 18.46 fixed point value. sqrtHigh is
    //---set to 0x000cfb72 and sqrtLow is set to 0x4a900000.
    UMUL64(*(NTO_U32 *) &n, rsq, &sqrtHigh, &sqrtLow);

    //---Determine the shift amount required to unnormalize the computed square root
    //---and convert the result to an I'.F' fixed point value. According to the
    //---section Fixed Point Implementation Notes On Computing Reciprocal Square
    //---Roots (see above), unnormalizing rsq requires shifting rsq to the right by
    //---(16 - 0.5 * (F + s)) bits. Furthermore, converting the 64-bit (2+I).(30+F)
    //---fixed point square root to an I'.F' fixed point value requires shifting the
    //---64-bit value to the right by (30 + F - F') bits. Therefore, the total shift
    //---amount required is (46 - F' + F/2 - s/2) = (shift - s/2) bits. Running
    //---example: n is a 16.16 fixed point value and the output is a 16.16 fixed
    //---point value, so F = F' = 16. Therefore, s = (shift - (s >> 1)) = (46 - F' +
    //---F/2 - s/2) = (46 - 16 + 16/2 - 6/2) = (30 + 8 - 3) = (38 - 3) = 35.
s = (shift - (s >> 1));

//----Compare the shift amount to 32
if (s >= 32) {

    //----The shift amount is at least 32. Therefore, it is sufficient to shift
    //----only the high 32 bits, because the low 32 bits are completely shifted
    //----off the right end. Running example: sqrt = (sqrtHigh >> (s - 32)) =
    //----(0x000fcfb72 >> (35 - 32)) = (0x00ff6e, which has
    //----the mathematical value (0x001f9f6e / 2^16) = 31.62277216796875. Note
    //----that the true value of the square root of 1000 is approximately
    //----31.62277660168379331998893544.
    sqrt = (sqrtHigh >> (s - 32));

} else {

    //----The shift amount is less than 32. Therefore, it is necessary to merge
    //----the high 32 bits and the low 32 bits of the 64-bit {2+I}.{30+F} fixed
    //----point value into a single 32-bit I'.F' fixed point value.
    sqrt = (sqrtHigh << (32 - s)) | (sqrtLow >> s);
}

//----Return the computed square root
return(sqrt);

//-----------------------------------------------------------------------------------
// Compute and return the positive square root of n, where both n and the computed
// result are non-negative NTO_I1616 fixed point values.
//
// Special cases are handled as follows. If n is zero, I1616_SQRT() returns exactly
// zero. If n is negative, the result is undefined.
//
// All assembly and C implementations produce bit-identical results.
//
// Implementation notes:
//
// - I1616_SQRT() computes the square root of n as:
//   
//   sqrt(n) = n^(1/2)
//   = n^(1 - 1/2)
//   = n * n^(-1/2)
//   = n * sqrt(n)
//
// where sqrt(n) denotes the reciprocal square root of n.
//
// - Although this approach can be implemented directly using the expression
//   I1616_MUL(n, I1616_RSQ(n)), the computed result is highly inaccurate because
//   14 intermediate fractional bits are lost when I1616_RSQ() converts the
//   reciprocal square root from its internal 2.30 fixed point representation to a
//   16.16 fixed point value (see above for more information about the I1616_RSQ()
//   function). Consequently, the computed product of n and I1616_RSQ(n) severely
//   underestimates the true product. This underestimate is significant when n is
//   large, because the reciprocal square root of a large number is a small number
//   (i.e., a value whose binary representation contains 1 bits in only the low
//   fractional bits).
//
// - To avoid this problem, I1616_SQRT() uses the high-precision SQRT() function
//   (see above) with the shift argument set to 38. Following the notation used in
//   the documentation for the SQRT() function, the input is a 16.16 fixed point
Implementation notes

- I2408_SQRT_I1616() computes the square root of n using the same approach used by the I1616_SQRT() function (see above). In this case, the shift argument (i.e., the second argument to the SQRT() function) is set to 34 instead of 38. Following the notation used in the documentation for the SQRT() function (see above), the input is a 24.8 fixed point value (i.e., I = 24 and F = 8) and the output is a 16.16 fixed point value (i.e., I' = F' = 16). Therefore shift = 46 - F' + F/2 = 46 - 16 + 8/2 = 30 + 4 = 34.

All assembly and C implementations produce bit-identical results.

Implementation notes:

1. Compute the reciprocal of d (i.e., 1/d). The reciprocal estimation method consists of a table lookup followed by two iterations of Newton-Raphson. The Newton-Raphson iteration equation to compute a reciprocal is x{i+1} = x{i} * (2 - d * x{i}), where x{i} is the estimate of the solution in the current iteration (i.e., iteration i) and x{i+1} is the computed estimate in the next iteration (i.e., iteration (i + 1)). See the above section Newton-Raphson Method Overview for an overview of the theory behind this numerical technique and for the derivation of the above formula.

A table lookup on the leading bits of the input d is used to obtain an initial approximation to the reciprocal of d. See the documentation for divTable[] below for more details about the table.

2. Multiply the reciprocal of d (i.e., 1/d) by n.
- The documentation for the C implementation below uses a running example to help explain the fixed point implementation. This running example considers the case where the input value n is the 0.32 fixed point value 0x40000000, which has the mathematical value (0x40000000 / 2^32) = 0.25, and the input value d is the 0.32 fixed point value 0x80000000, which has the mathematical value (0x80000000 / 2^32) = 0.5. The true quotient (n / d) is 0.5.

Performance notes:

Intel Centrino Core Duo T2500 (2 MB L2, 2.0 GHz, FSB 677 MHz), MSVC 6 compiler,

Release mode:

- NTO_MATH_FIXED_C_64 is ~1.3x as fast as NTO_MATH_FIXED_C_32
- NTO_MATH_FIXED_ASM_X86 is ~1.6x as fast as NTO_MATH_FIXED_C_64
- NTO_MATH_FIXED_ASM_X86 is ~2.0x as fast as NTO_MATH_FIXED_C_32

divTable[] is the reciprocal lookup table used by the DIV() function. The table contains 128 elements and is precomputed as follows:

divTable[i] = round(65536.0 / (i + 128.5)) - 256, where i is an integer that lies in the range [0, 127]. The following discussion explains this formula.

The purpose of divTable[] is to provide a fast and reasonably accurate initial estimate to the reciprocal of a number d.

Assume that d is a 0.32 fixed point value that lies in the mathematical range (0.5, 1). Considered as an unsigned integer, d is greater than 2^31. Therefore, the MSB of d is 1.

The reciprocal of each value in the range (0.5, 1) must lie in the range (1, 2). Therefore, the reciprocal of each value in the range (0.5, 1) can be expressed as a 24.8 fixed point value, where the integer portion of the 24.8 fixed point value is always 1. For example, the reciprocal of 0.7 can be expressed as the 24.8 fixed point value 0x16e, which has the mathematical value (0x16e / 2^8) = 1.4296875. Consequently, the table only needs to store the low 8 bits (i.e., the fractional bits) of the reciprocal of each value in the range (0.5, 1). Storing the integer portion of the 24.8 fixed point value in the table is unnecessary because it is always 1.

For a good tradeoff between table size and the accuracy of the initial estimate, this implementation uses the seven fractional bits of d following the leading 1 to index into the table, thereby limiting the table size to 128 elements. Note that the MSB of d is not used to index into the table because it is always 1.

Computing the table is accomplished with the following steps:

Let i be the 7-bit integer index formed by the seven bits following the leading 1 of d (i.e., bits 30:24 of d).

First, map an integer index i in the range [0, 127] to the range [0.501953125, 0.998046875], which is an approximation to the range (0.5, 1):

(i + 128.5) / 256.0

The value of 128.5 (instead of 128.0) is chosen for technical reasons explained below. Next, compute the reciprocal:

256.0 / (i + 128.5)

Scale the result by 256.0 and round to the nearest integer to obtain a 24.8 fixed point value:

round(65536.0 / (i + 128.5))

Finally, subtract (i.e., remove) the integer portion of the 24.8 fixed point value because the integer portion is always 1. Putting all the steps together yields the following formula:
The value 256 requires 9 bits to store. By choosing a value slightly larger than 128.0, such as 128.5, the following result is obtained:

```
rsqTable[0] = round(65536.0 / (0 + 128.5)) - 256
```

Although the above discussion has assumed that d lies in the mathematical range (0.5, 1), the DIV() function uses divTable[] to compute reciprocals of values that lie in the mathematical range [0.5, 1) (note the inclusion of 0.5). The first element of divTable[] (i.e., divTable[0]) is an estimate of the reciprocal of 0.5.

```
static const NTO_U8 divTable[] = {
    0x0f, 0xf0, 0xf6, 0xf2, 0xef, 0xe0b, 0xe7, 0xe4,
    0xe0, 0x0dd, 0xd9, 0xd6, 0xd2, 0xcf, 0xcc, 0xc9,
    0xc6, 0xc2, 0xbf, 0xbc, 0xb9, 0xb6, 0xb3, 0xb1,
    0xae, 0xab, 0xa8, 0xa5, 0xa3, 0xa0, 0x9d, 0x9b,
    0x98, 0x96, 0x93, 0x91, 0x8e, 0x8c, 0x8a, 0x87,
    0x85, 0x83, 0x80, 0x7e, 0x7c, 0x7a, 0x78, 0x75,
    0x73, 0x71, 0x6f, 0x6d, 0x6b, 0x69, 0x67, 0x65,
    0x63, 0x61, 0x5f, 0x5e, 0x5c, 0x5a, 0x58, 0x56,
    0x54, 0x53, 0x51, 0x4f, 0x4e, 0x4c, 0x4a, 0x49,
    0x47, 0x45, 0x44, 0x42, 0x40, 0x3f, 0x3d, 0x3c,
    0x3a, 0x39, 0x37, 0x36, 0x34, 0x33, 0x32, 0x30,
    0x2f, 0x2d, 0x2c, 0x2b, 0x29, 0x28, 0x27, 0x25,
    0x24, 0x23, 0x21, 0x20, 0x1f, 0x1e, 0x1c, 0x1b,
    0x1a, 0x19, 0x17, 0x16, 0x15, 0x14, 0x13, 0x12,
    0x10, 0x0f, 0xe0, 0xe0d, 0xe0c, 0xe0b, 0xe0a, 0xe09,
    0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
};`
C:\\Users\\perry\\Desktop\\MiniNitro\\Nitro\\FixedMath.h

//---portion of the 24.8 fixed point estimate (i.e., 0x100), which always
//---has the mathematical value 1. The result is a 24.8 fixed point estimate
//---to 1/d that lies in the mathematical range (1, 2). Running example:
//---consider the case where d = 0x80000000 (i.e., the mathematical value
//----(0x80000000 / 2^32) = 0.5). x0 = divTable[(d >> 24) - 128] + 0x100 =
//----divTable[(0x80000000 >> 24) - 128] + 0x100 = divTable[128 - 128] +
//----0x100 = divTable[0] + 0x100 = 0xfe + 0x100 = 0x1fe, which has the
//---mathematical value (0x1fe / 2^8) = 1.9921875. This is the initial
//---estimate of 1/0.5.
//x0 = divTable[(d >> 24) - 128] + 0x100;

//---Begin the first Newton-Raphson iteration. Compute the 16.16 fixed point
//---value (x0 * x0). Note that x0 is a 24.8 fixed point value that lies in
//---the mathematical range (1, 2), so the product (x0 * x0) is a 64-bit
//---16.16 fixed point value whose high 32 bits are all zero. Therefore, it
//---suffices to keep only the low 32 bits of the 64-bit product,
//---effectively converting the 48.16 fixed point value to a 16.16 fixed
//---point value. Running example: x1 = x0 * x0 = 0x1fe * 0x1fe = 0x3f804,
//---which has the mathematical value (0x3f804 / 2^16) = 3.96881103515625.
x1 = x0 * x0;

//---Compute the 16.16 fixed point value (d * x0 * x0). Note that d is a
//---0.32 fixed point value and x1 is a 16.16 fixed point value, so the
//---product of d and x1 is a 64-bit 16.48 fixed point value. Keeping only
//---the high 32 bits of the 64-bit product effectively truncates the 16.48
//---fixed point value to a 16.16 fixed point value. Running example: d * x1
//-----0x80000000 * 0x3f804 = 0x1fc0200000000. Keeping only the high 32 bits
//-----yields x1 = 0x1fc02, which has the mathematical value (0x1fc02 / 2^16)
//-----1.984405517578125.
x1 = UMUL64_HIGH32(d, x1);

//---Convert x0 from a 24.8 fixed point value to a 16.16 fixed point value
//---(i.e., shift x0 to the left by 8 bits) and multiply x0 by two (i.e.,
//---shift x0 to the left by an additional bit). Then compute the 16.16
//---fixed point value ((2 * x0) - (d * x0 * x0)). This is the end of the
//---first Newton-Raphson iteration. Running example: x1 = (x0 << 9) - x1 =
//-----((0x1fe << 9) - 0x1fc02 = 0x3fc00 - 0x1fc02 = 0x1ffe, which has
//-----the mathematical value (0x1ffe / 2^16) = 1.999969482421875. Note that
//-----x1 is a better estimate of 1/0.5 than x0.
x1 = (x0 << 9) - x1;

//---Begin the second Newton-Raphson iteration. Compute the 64-bit 32.32
//---fixed point value (x1 * x1). Set x2High to the high 32 bits of the
//---64-bit product and set x2Low to the low 32 bits of the 64-bit product.
//---Note that x1 is a 16.16 fixed point estimate to 1/d that lies in the
//---mathematical range (1, 2). Therefore, the high 30 bits of x2High are
//---zero but at least one of the low two bits of x2High is 1. Running
//---example: x1 * x1 = 0x3fff80004, which has the mathematical value
//-----(0x3fff80004 / 2^32) = 3.999877930618822574615478515625. x2High = 0x3
//-----and x2Low = 0x8ff80004. Note that the low two bits of x2High are both
//-----1.
x1 = UMUL64(x1, x1, &x2High, &x2Low);

//---Convert the 32.32 fixed point value (x1 * x1) to a 33.31 fixed point
//---value by shifting (x1 * x1) to the right by 1 bit and rounding the
//---result. Note that because the variable x2 can only store 32 bits, bit 1
//---of x2High is not stored in x2. The potential error due to this lost bit
//---is corrected in a separate step below. Running example: x2 = (x2Low >>
//------1) + (x2High << 31) + (x2Low & 1) = (0x8ff80004 >> 1) + (0x3 << 31) +
//------(0x8ff80004 & 1) = (0x7ffcc0002) + (0x180000000) + (0) = 0x1ffcc0002.
//---Note that this value is stored as 0x1ffcc0002 (i.e., the high bit is
//-----lost).
x2 = (x2Low >> 1) + (x2High << 31) + (x2Low & 1);

//-----Compute the 1.31 fixed point value (d * x1 * x1). Note that d is a 0.32
//-----fixed point value and x2 is a 1.31 fixed point value, so the product of
//-----d and x1 is a 64-bit 1.63 fixed point value. Keeping only the high 32
//-----bits of the 64-bit product effectively truncates the result to a 1.31
//-----fixed point value. Running example: d * x2 = 0x80000000 * 0xffffffff =
//-----0xffe0000100000000. Keeping only the high 32 bits yields x2 =
//-----0x7ffe0001, which has the mathematical value (0x7ffe0001 / 2^31) =
//-----0.9993896539412873077392578125.
//-----Recall that bit 1 of x2High was lost when converting the 32.32 fixed
//-----point value (x1 * x1) to a 33.31 fixed point value and storing the
//-----result into a 32-bit integer (see above). If bit 1 of x2High is 1, then
//-----correct for this error by adding the 1.31 fixed point value (2 * d) to
//-----the 1.31 fixed point value x2. Since d is a 0.32 fixed point value, it
//-----is already the desired 1.31 fixed point value (2 * d). Therefore,
//-----simply add d to x2. Running example: x2High is 0x3, so bit 1 of x2High
//-----is 1. Therefore, x2 is set to x2 + d = 0x7ffe0001 + 0x80000000 =
//-----0xffe0001, which has the mathematical value (0xffe0001 / 2^31) =
//-----1.9993896539412873077392578125.
//-----Compute the 1.31 fixed point value ((2 * x1) - (d * x1 * x1)). Note
//-----that shifting x1 to the left by 15 bits converts x1 from a 16.16 fixed
//-----point value to a 1.31 fixed point value. Shifting x1 to the left by an
//-----additional bit effectively multiplies x1 by 2. This is the end of the
//-----second Newton-Raphson iteration. Running example: x2 = (x1 << 16) - x2
//-----((0x1fffffff - 0xfffff000) - 0xffffffff) = 0xffffffff, which has the mathematical value (0xffffffff / 2^32) =
//-----1.9999999995343387126922607421875. Note that x2 is a better estimate of
//-----1/0.5 than x1. The true value of 1/0.5 is 2.
//-----Compute return the 1.31 fixed point value n/d. Evaluate n/d by
//-----multiplying 1/d (i.e., x2) by n. Note that x2 is a 1.31 fixed point
//-----value and n is a 0.32 fixed point value, so their product is a 64-bit
//-----fixed point value. Keeping only the high 32 bits of the 64-bit
//-----product effectively truncates the result to a 1.31 fixed point value.
//-----Since the leading 1 of x2 lies at bit position 31 (i.e., the MSB) and
//-----the leading 1 of n lies at bit position 30, the leading 1 of the
//-----resulting 1.31 fixed point value n/d lies at bit position 30. Running
//-----example: consider the case where n is the 0.32 fixed point value
//-----0x400000000, which has the mathematical value (0x400000000 / 2^32) =
//-----0.25. Then n * x2 = 0x400000000 * 0xffffffff = 0x3fffffffc00000000.
//-----Keeping only the high 32 bits yields the result 0x3fffffff, which has
//-----the mathematical value (0x3fffffff / 2^31) =
//-----0.4999999995343387126922607421875. Note that the true value of n/d in
//-----this example is 0.25/0.5 = 0.5.
return(UMUL64_HIGH32(n, x2));

#elf (NTO_MATH_MODE == NTO_MATH_FIXED_ASM_X86)
{

//x86 assembly implementation

asm

//-----Set ecx and edi to the denominator d
mov ecx, d;
mov edi, d;
//-----Note that the MSB of d is 1. Perform a table lookup using the seven
//-----bits of d following the MSB (i.e., bits 30:24) to obtain the 8
//-----fractional bits of a 24.8 fixed point estimate to 1/d. Add the
//-----integer portion of the 24.8 fixed point estimate (i.e., 0x100),
//-----which always has the mathematical value 1. The result is a 24.8
//-----fixed point estimate to 1/d that lies in the mathematical range (1,
//-----2).
shr   edi, 24;
sub   edi, 128;
mov   eax, 0;
mov   al, divTable[edi];
add   eax, 256;

//-----Let x0 be the 24.8 fixed point estimate to 1/d (i.e., eax). Copy x0
//-----to ebx.
mov   ebx, eax;

//-----Begin the first Newton-Raphson iteration. Compute the 16.16 fixed
//-----point value [edx:eax] = (x0 * x0). Note that x0 is a 24.8 fixed
//-----point value that lies in the mathematical range (1, 2), so the
//-----product (x0 * x0) is a 64-bit 48.16 fixed point value whose high 32
//-----bits are all zero. Therefore, it suffices to keep only the low 32
//-----bits of the 64-bit product, effectively converting the 48.16 fixed
//-----point value to a 16.16 fixed point value.
imul  eax, eax;

//-----Compute the 16.16 fixed point value edx = (d * x0 * x0). Note that
//-----d is a 0.32 fixed point value and x0 is a 16.16 fixed point value,
//-----so the product of d and x0 is a 64-bit 16.48 fixed point value.
//-----Keeping only the high 32 bits of the 64-bit product effectively
//-----truncates the 16.48 fixed point value to a 16.16 fixed point value.
mul   ecx;

//-----Convert x0 from a 24.8 fixed point value to a 16.16 fixed point
//-----value (i.e., shift x0 to the left by 8 bits) and multiply x0 by two
//-----(i.e., shift x0 to the left by an additional bit). Then compute the
//-----16.16 fixed point value ((2 * x0) - (d * x0 * x0)). This is the end
//-----of the first Newton-Raphson iteration.
shl   ebx, 9;
sub   ebx, edx;

//-----Begin the second Newton-Raphson iteration. Compute the 64-bit 32.32
//-----fixed point value [edx:eax] = (x1 * x1). Set edx to the high 32
//-----bits of the 64-bit product and set eax to the low 32 bits of the
//-----64-bit product. Note that x1 is a 16.16 fixed point estimate to 1/d
//-----that lies in the mathematical range (1, 2). Therefore, the high 30
//-----bits of edx are zero but at least one of the low two bits of edx is
//-----1.
mov   eax, ebx;
mul   eax;

//-----Convert the 32.32 fixed point value (x1 * x1) to a 33.31 fixed
//-----point value by shifting (x1 * x1) to the right by 1 bit and
//-----rounding the result. Note that because the register eax can only
//-----store 32 bits, bit 1 of edx is not stored in eax. The potential
//-----error due to this lost bit is corrected in a separate step below.
mov   edi, edx;
shl   edx, 31;
shr   eax, 1;
adc   eax, edx;
/-----Compute the 1.31 fixed point value edx = (d * x1 * x1). Note that d
/-----is a 0.32 fixed point value and x2 is a 1.31 fixed point value, so
/-----the product of d and x1 is a 64-bit 1.63 fixed point value. Keeping
/-----only the high 32 bits of the 64-bit product effectively truncates
/-----the result to a 1.31 fixed point value.
mul ecx;

//-----Recall that bit 1 of edx was lost when converting the 32.32 fixed
//-----point value (x1 * x1) to a 33.31 fixed point value and storing the
//-----result into a 32-bit integer (see above). If bit 1 of edi is 1,
//-----then correct for this error by adding the 1.31 fixed point value (2
//-----* d) to the 1.31 fixed point value x2. Since d is a 0.32 fixed
//-----point value, it is already the desired 1.31 fixed point value (2 *
//-----d). Therefore, simply add d to x2.
shl edi, 30;
jns end;
add edx, ecx;

//-----Compute the 1.31 fixed point value ((2 * x1) - (d * x1 * x1)). Note
//-----that shifting x1 to the left by 15 bits converts x1 from a 16.16
//-----fixed point value to a 1.31 fixed point value. Shifting x1 to the
//-----left by an additional bit effectively multiplies x1 by 2. This is
//-----the end of the second Newton-Raphson iteration.
end:
shl ebx, 16;
sub ebx, edx;

//-----Compute and return the 1.31 fixed point value n/d. Evaluate n/d by
//-----multiplying 1/d (i.e., x2) by n. Note that x2 is a 1.31 fixed point
//-----value and n is a 0.32 fixed point value, so their product is a
//-----64-bit 1.63 fixed point value. Keeping only the high 32 bits of the
//-----64-bit product effectively truncates the result to a 1.31 fixed
//-----point value. Since the leading 1 of x2 lies at bit position 31
//-----(i.e., the MSB) and the leading 1 of n lies at bit position 30, the
//-----leading 1 of the resulting 1.31 fixed point value n/d lies at bit
//-----position 30.
mov eax, n;
mul ebx;

//-----Return the quotient
mov eax, edx;

} #endif

/***********************************************************************
* Compute and return the signed quotient (n / d). The input numerator n, the input
* denominator d, and the computed quotient are NTO_I1616 fixed point values. The
* quotient is rounded towards zero.
* On output, I1616_DIV() sets status to NTO_FIXED_MATH_NO_ERROR,
* NTO_FIXED_MATH_OVERFLOW, or NTO_FIXED_MATH_UNDERFLOW, depending on the outcome of the quotient computation. The possible cases are as
* follows:
* 1. If n is any value and d is zero, I1616_DIV() returns zero and sets status to
* NTO_FIXED_MATH_NAN.
* 2. If n is zero and d is non-zero, I1616_DIV() returns zero and sets status to
* NTO_FIXED_MATH_NO_ERROR.
3. If \( n \) is non-zero and \( d \) is 0x10000 (i.e., the mathematical value 1), \n\[ \text{IL166}_\text{DIV}() \text{ returns } n \text{ and sets status to NTO\_FIXED\_MATH\_NO\_ERROR}. \]

4. If \( n \) is non-zero and \( d \) is 0xffffffff (i.e., the mathematical value -1), \n\[ \text{IL166}_\text{DIV}() \text{ returns } -n \text{ and sets status to NTO\_FIXED\_MATH\_NO\_ERROR}. \]

5. If \( n \) and \( d \) are both non-zero and the quotient \( (n / d) \) overflows the \n\[ \text{NTO}_{\text{IL166}} \text{ fixed point representation, IL166}_\text{DIV}() \text{ returns zero and sets} \]
\[ \text{status to NTO\_FIXED\_MATH\_OVERFLOW}. \]

6. If \( n \) and \( d \) are both non-zero and the quotient \( (n / d) \) underflows the \n\[ \text{NTO}_{\text{IL166}} \text{ fixed point representation, IL166}_\text{DIV}() \text{ returns zero and sets} \]
\[ \text{status to NTO\_FIXED\_MATH\_UNDERFLOW}. \]

7. In all other cases, IL166_DIV() computes and returns the signed fixed point \n\[ \text{quotient } (n / d) \text{ and sets status to NTO\_FIXED\_MATH\_NO\_ERROR}. \]

All assembly and C implementations produce bit-identical results.

Implementation notes:

- IL166_DIV() computes the quotient of \( n \) and \( d \) using the following steps:
  1. Normalize abs\((n)\) to a 0.32 fixed point value that lies in the \n  mathematical range \([0.25, 0.5)\). Normalize abs\((d)\) to a 0.32 fixed point \n  value that lies in the mathematical range \([0.5, 1)\). \n  2. Compute the unsigned quotient of the normalized values of abs\((n)\) and \n  abs\((d)\) by calling the DIV() function (see above). \n  3. Unnormalize the unsigned quotient and convert the result to a signed \n  16.16 fixed point value.

Performance notes:

- Intel Centrino Core Duo T2500 (2 MB L2, 2.0 GHz, FSB 677 MHz): MSVC 6 compiler, \n  Release mode:
    - NTO\_MATH\_FIXED\_C\_64 is \( \sim 1.2x \) as fast as NTO\_MATH\_FIXED\_C\_32
    - NTO\_MATH\_FIXED\_ASM\_X86 is \( \sim 1.4x \) as fast as NTO\_MATH\_FIXED\_C\_64
    - NTO\_MATH\_FIXED\_ASM\_X86 is \( \sim 1.7x \) as fast as NTO\_MATH\_FIXED\_C\_32

NTO_IL166_LONG16 IL166_DIV (NTO_IL166 n, NTO_IL166 d, NTO_I32 *status)

\[ \text{NTO} \_\text{I32} \; s; \]
\[ \text{NTO} \_\text{I32} \; q; \]
\[ \text{NTO} \_\text{I32} \; n\text{Bits}, \; d\text{Bits}; \]
\[ \text{NTO} \_\text{I32} \; d\text{Shift}, \; n\text{Shift}; \]
\[ \text{NTO} \_\text{I32} \; n\text{Sign}, \; d\text{Sign}, \; q\text{Sign}; \]

//-----Initialize status to FIXED\_MATH\_NO\_ERROR (i.e., no error has occurred) \n*status = NTO\_FIXED\_MATH\_NO\_ERROR;

//-----If the denominator \( d \) is zero, set status to NTO\_FIXED\_MATH\_NAN and return \n//-----zero
if (!d) {
  *status = NTO\_FIXED\_MATH\_NAN;
  return(0);
}

//-----If the numerator is zero, return zero
if (!n) return(0);
//----If d is 0x10000 (i.e., the mathematical value 1), return n
if (d == 0x10000) return(n);

//----If d is 0xffffffff (i.e., the mathematical value -1), return -n
if (d == 0xffffffff) return(-n);

//----Extract the signs of n and d
nSign = n >> 31;
dSign = d >> 31;

//----Compute the sign of the quotient
qSign = nSign ^ dSign;

//----Set nBits to abs(n) and set dBits to abs(d)
nBits = (n < 0) ? -n : n;
dBits = (d < 0) ? -d : d;

//----Compute the integer exponent dShift required to convert dBits from a 16.16
//----fixed point value to a 0.32 fixed point value that is normalized to the
//----mathematical range [0.5, 1) (i.e., determine dShift such that 0.5 <=
//----((dBits * 2^dShift) / 2^32) < 1).
dShift = CountLeadingZeroes(dBits);

//----Compute the integer exponent nShift required to convert nBits from a 16.16
//----fixed point value to a 0.32 fixed point value that is normalized to the
//----mathematical range [0.25, 0.5) (i.e., determine nShift such that 0.25 <=
//----((nBits * 2^nShift) / 2^32) < 0.5).
nShift = CountLeadingZeroes(nBits) - 1;

//----Normalize dBits to a 0.32 fixed point value that lies in the mathematical
//----range [0.5, 1)
dBits <<= dShift;

//----Normalize nBits to a 0.32 fixed point value that lies in the mathematical
//----range [0.25, 0.5)
nBits <<= nShift;

//----Determine the shift amount required to unnormlize the 1.31 fixed point
//----value q and convert the result to a 16.16 fixed point value. Unnormalizing
//----q requires shifting q to the right by (nShift - dShift) bits. To understand
//----this formula, recall that nBits = (n * 2^nShift) and that dBits = (d *
//----2^dShift) (see above). Therefore q = (nBits / dBits) = (n * 2^nShift) / (d
//----* 2^dShift) = (n / d) * (2^nShift / 2^dShift) = (n / d) * 2^(nShift -
//----dShift). It follows that (n / d) = (nBits / dBits) * 2^(dShift - nShift) =
//----q * 2^(dShift - nShift). Therefore, unnormalizing q requires multiplying q
//----by 2^(dShift - nShift), which is equivalent to shifting q to the right by
//----(nShift - dShift) bits. Converting q from a 1.31 fixed point value to a
//----16.16 fixed point value requires shifting q to the right by 15 bits.
//----Therefore, q must be shifted to the right by a total of (15 + nShift -
//----dShift) bits.
s = 15 + nShift - dShift;

//----Determine if s is less than -31 or greater than 31
if (s < -31) {

//----The required shift amount is less than -31 (i.e., the quotient must be
//----shifted left by at least 32 bits, thereby overflowing the NTO_I1616
//----fixed point representation). Set status to NTO_FIXED_MATH_OVERFLOW and
//----return zero.
*status = NTO_FIXED_MATH_OVERFLOW;
return(0);
}

} else if (s > 31) {

//----The required shift amount is greater than 31 (i.e., the quotient must
//----be shifted right by at least 32 bits, thereby underflowing the
//----NTO_I1616 fixed point representation). Set status to
//----NTO_FIXED_MATH_UNDERFLOW and return zero.
*status = NTO_FIXED_MATH_UNDERFLOW;
return(0);
}

//----Compute the unsigned quotient of nBits and dBits (i.e., nBits/dBits). Note
//----that the DIV() function computes the result as a 1.31 fixed point value
//----whose leading 1 bit is at bit position 30.
q = DIV(nBits, dBits);

//----Determine if s is negative or positive
if (s < 0) {

//----The required shift amount is negative and lies in the range [-31, -1].
//----Therefore, unnormalizing q requires shifting q to the left by -s bits.
//----Determine if the unnormalized quotient (i.e., q << (-s)) overflows the
//----NTO_I1616 fixed point representation.
if ((q >> (32 + s)) != 0) {

//----At least one of the abs(s) most significant bits of q is a 1 bit.
//----This 1 bit is lost when shifting q to the left by -s bits.
//----Therefore, the unnormalized quotient (i.e., q << (-s)) overflows
//----the NTO_I1616 fixed point representation. Set status to
//----NTO_FIXED_MATH_OVERFLOW and return zero.
*status = NTO_FIXED_MATH_OVERFLOW;
return(0);
}

//----The unnormalized quotient does not overflow the NTO_I1616 fixed point
//----representation. Shift q to the left by -s bits.
q <<= -s;
}

} else {

//----The required shift amount is non-negative and lies in the range [0,
//----31]. Shift q to the right by s bits.
q >>= s;

//----If the normalized quotient (i.e., q) is zero, then the quotient
//----underflows the NTO_I1616 fixed point representation. Set status to
//----NTO_FIXED_MATH_UNDERFLOW and return zero.
if (q == 0) {
*status = NTO_FIXED_MATH_UNDERFLOW;
return(0);
}
}
The unnormalized quotient neither overflows nor underflows the NTO_I1616 fixed point representation. If the quotient is non-negative, return the quotient. If the quotient is negative, negate the unsigned quotient and return the result. This step has the effect of rounding the quotient towards zero.

```c
return(qSign ? -((NTO_I32) q) : q);
```