```
#ifndef _FPTC_H_
#define _FPTC_H_

/*
 * fptc.h is a 32-bit or 64-bit fixed point numeric library (modified by mgetka)
 *
 * The symbol FPT_BITS, if defined before this library header file
 * is included, determines the number of bits in the data type (its "width").
 * The default width is 32-bit (FPT_BITS=32) and it can be used
 * on any recent C99 compiler. The 64-bit precision (FPT_BITS=64) is
 * available on compilers which implement 128-bit "long long" types. This
 * precision has been tested on GCC 4.2+.
 *
 * The FPT_WBITS symbols governs how many bits are dedicated to the
 * "whole" part of the number (to the left of the decimal point). The larger
 * this width is, the larger the numbers which can be stored in the fpt
 * number. The rest of the bits (available in the FPT_FBITS symbol) are
 * dedicated to the fraction part of the number (to the right of the decimal
 * point).
 *
 * Since the number of bits in both cases is relatively low, many complex
 * functions (more complex than div & mul) take a large hit on the precision
 * of the end result because errors in precision accumulate.
 * This loss of precision can be lessened by increasing the number of
 * bits dedicated to the fraction part, but at the loss of range.
 *
 * Adventurous users might utilize this library to build two data types:
 * one which has the range, and one which has the precision, and carefully
 * convert between them (including adding two number of each type to produce
 * a simulated type with a larger range and precision).
 *
 * The ideas and algorithms have been cherry-picked from a large number
 * of previous implementations available on the Internet.
 * Tim Hartrick has contributed cleanup and 64-bit support patches.
 *
 * == Special notes for the 32-bit precision ==
 * Signed 32-bit fixed point numeric library for the 24.8 format.
 * The specific limits are -8388608.999... to 8388607.999... and the
 * most precise number is 0.00390625. In practice, you should not count
 * on working with numbers larger than a million or to the precision
 * of more than 2 decimal places. Make peace with the fact that PI
 * is 3.14 here. :)
 */

 /*
 * Copyright (c) 2017 Michał Getka <michal.getka@gmail.com>
 * This file contains modified version of original fixed point library. Some
 * additional functionalities and improvements were made.
 */

/*-
 * Copyright (c) 2010-2012 Ivan Voras <ivoras@freebsd.org>
 * Copyright (c) 2012 Tim Hartrick <tim@edgecast.com>
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
```

```
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

#ifndef FPT_BITS
#define FPT_BITS  32
#endif

#include <stdint.h>

#if FPT_BITS == 32
typedef int32_t fpt;
typedef int64_t  fptd;
typedef uint32_t fptu;
typedef uint64_t fptud;
#elif FPT_BITS == 64
typedef int64_t fpt;
typedef __int128_t fptd;
typedef uint64_t fptu;
typedef __uint128_t fptud;
#else
#error "FPT_BITS must be equal to 32 or 64"
#endif

#ifndef FPT_WBITS
#define FPT_WBITS  17
#endif

#if FPT_WBITS >= FPT_BITS
#error "FPT_WBITS must be less than or equal to FPT_BITS"
#endif

#define FPT_VCSID "$Id$"

#define FPT_FBITS  (FPT_BITS - FPT_WBITS)
#define FPT_FMASK  (((fpt)1 << FPT_FBITS) - 1)

#define fl2fpt(R) ((fpt)((R) * FPT_ONE + ((R) >= 0 ? 0.5 : -0.5)))
#define i2fpt(I) ((fptd)(I) << FPT_FBITS)
#define fpt2i(F) ((F) >> FPT_FBITS)

#define i2fpt_norm(I,n) (               \
    (FPT_FBITS - n) >= 0 ?              \
      ((fptd)(I) << (FPT_FBITS - n)) :  \
      ((fptd)(I) >> -(FPT_FBITS - n))   \
  )
```

```
#define fpt2i_norm(F,n) (               \
    (FPT_FBITS - n) >= 0 ?              \
      ((F) >> (FPT_FBITS - n)) :        \
      ((F) << -(FPT_FBITS - n))         \
  )
#define fpt_norm(F,from,to) (           \
    (from - to) >= 0 ?                  \
      ((fptd)(F) << (from - to)) :      \
      ((fptd)(F) >> -(from - to))       \
  )                                     \

#define fpt_xadd(A,B) ((A) + (B))
#define fpt_xsub(A,B) ((A) - (B))
#define fpt_xmul(A,B)                   \
  ((fpt)(((fptd)(A) * (fptd)(B)) >> FPT_FBITS))
#define fpt_xdiv(A,B)                   \
  ((fpt)(((fptd)(A) << FPT_FBITS) / (fptd)(B)))
#define fpt_fracpart(A) ((fpt)(A) & FPT_FMASK)

#define FPT_ONE       ((fpt)((fpt)1 << FPT_FBITS))
#define FPT_ZERO      ((fpt)0)
#define FPT_MINUS_ONE (-FPT_ONE)
#define FPT_ONE_HALF  (FPT_ONE >> 1)
#define FPT_TWO       (FPT_ONE + FPT_ONE)
#define FPT_MAX       ((fpt)((fptu)~0 >> 1))
#define FPT_MIN       (~FPT_MAX)
#define FPT_ABS_MAX   FPT_MAX
#define FPT_ABS_MIN   ((fpt)1)
#define FPT_PI        fl2fpt(3.14159265358979323846)
#define FPT_TWO_PI    fl2fpt(2 * 3.14159265358979323846)
#define FPT_HALF_PI   fl2fpt(3.14159265358979323846 / 2)
#define FPT_E         fl2fpt(2.7182818284590452354)

/* Following block of code defines default overflow handling macros for math
 * operations. */

#ifndef _fpt_add_overflow_handler
#define _fpt_add_overflow_handler return FPT_MAX;
#endif
#ifndef _fpt_add_underflow_handler
#define _fpt_add_underflow_handler return FPT_MIN;
#endif

#ifndef _fpt_sub_overflow_handler
#define _fpt_sub_overflow_handler return FPT_MAX;
#endif
#ifndef _fpt_sub_underflow_handler
#define _fpt_sub_underflow_handler return FPT_MIN;
#endif

#ifndef _fpt_mul_overflow_handler
#define _fpt_mul_overflow_handler return FPT_MAX;
#endif
#ifndef _fpt_mul_underflow_handler
#define _fpt_mul_underflow_handler return FPT_MIN;
#endif

#ifndef _fpt_div_overflow_handler
#define _fpt_div_overflow_handler return FPT_MAX;
```

```
#endif
#ifndef _fpt_div_underflow_handler
#define _fpt_div_underflow_handler return FPT_MIN;
#endif

#define fpt_abs(A) ((A) < 0 ? -(A) : (A))

/* fpt is meant to be usable in environments without floating point support
 * (e.g. microcontrollers, kernels), so we can't use floating point types directly.
 * However floats can occur in places that will be optimised out during
compilation. */
#define fpt2fl(T) ((float) ((T)*((float)(1)/(float)(1 << FPT_FBITS))))

/* Adds two fpt numbers, returns the result. */
//static inline fpt
fpt fpt_add(fpt A, fpt B)
{

  #ifdef FPT_ADD_OVERFLOW_HANDLING
  if ((A > 0) && (B > FPT_MAX - A)) _fpt_add_overflow_handler;
  if ((A < 0) && (B < FPT_MIN - A)) _fpt_add_underflow_handler;
  #endif

  return ((A) + (B));
}


/* Subtracts two fpt numbers, returns the result. */
//static inline fpt
fpt fpt_sub(fpt A, fpt B)
{

  #ifdef FPT_SUB_OVERFLOW_HANDLING
  if ((A < 0) && (B > FPT_MAX + A)) _fpt_sub_overflow_handler;
  if ((A > 0) && (B < FPT_MIN + A)) _fpt_sub_underflow_handler;
  #endif

  return ((A) - (B));
}


/* Multiplies two fpt numbers, returns the result. */
//static inline fpt
fpt fpt_mul(fpt A, fpt B)
{

  #ifdef FPT_MUL_OVERFLOW_HANDLING
  if (A < FPT_ZERO && B < FPT_ZERO) {
    if ((fptd)A < ((fptd)FPT_MAX << FPT_FBITS) / (fptd)B)
_fpt_mul_overflow_handler;
  } else if (B > FPT_ZERO) {
    if ((fptd)A > ((fptd)FPT_MAX << FPT_FBITS) / (fptd)B)
_fpt_mul_overflow_handler;
    if ((fptd)A < ((fptd)FPT_MIN << FPT_FBITS) / (fptd)B)
_fpt_mul_underflow_handler;
  } else if (B < FPT_ZERO) {
    if ((fptd)A > ((fptd)FPT_MIN << FPT_FBITS) / (fptd)B)
_fpt_mul_underflow_handler;
  }
```

```
  #endif

  return (((fptd)A * (fptd)B) >> FPT_FBITS);
}


/* Divides two fpt numbers, returns the result. */
//static inline fpt
fpt fpt_div(fpt A, fpt B)
{

  #ifdef FPT_DIV_OVERFLOW_HANDLING
  /* The purpose of overflow handling mechanism is not to detect zero division.
   * So it is not checked if A is zero. If this is possible, it should be
   * handled outside the library, and if it is, it would generate excessive
   * computation overhead when doing this in here.*/
  if (fpt_abs(B) <= FPT_ONE) {
    if (A < FPT_ZERO && B < FPT_ZERO) {
      if ((fptd)A << FPT_FBITS < (fptd)FPT_MAX * (fptd)B)
_fpt_div_overflow_handler;
    } else if (B > FPT_ZERO) {
      if ((fptd)A << FPT_FBITS > (fptd)FPT_MAX * (fptd)B)
_fpt_div_overflow_handler;
      if ((fptd)A << FPT_FBITS < (fptd)FPT_MIN * (fptd)B)
_fpt_div_underflow_handler;
    } else if (B < FPT_ZERO) {
      if ((fptd)A << FPT_FBITS > (fptd)FPT_MIN * (fptd)B)
_fpt_div_underflow_handler;
    }
  }
  #endif

  return (((fptd)A << FPT_FBITS) / (fptd)B);
}

/*
 * Note: adding and substracting fpt numbers can be done by using
 * the regular integer operators + and -.
 */

//static inline int
int _pow(int x, unsigned int y) {

  unsigned int i;
  int ret = 1;

  for (i = 0; i<y; i++)
    ret *= x;

  return ret;
}

/* Parse string to fpt. Scientific format is not supported. */
//static inline int
int fpt_scan(const char * s, fpt * num, int * br) {

  unsigned int i = 0;
  int ret = 0;
  int whole = 0;
```

```
  int decimal = 0;
  int expo = 0;
  int bytesread = 0;
  int neg = 0;

  do {

    if (*(s+i) == '-') {
      neg = 1;
      i += 1;
      ret += 1;
    }

    if (*(s+i) >= '0' && *(s+i) <= '9') {
      if (sscanf(s+i, "%d%n", &whole, &bytesread) > 0)
        ret = 1;
      i += bytesread;
    }

    if (*(s+i) == '.' && *(s+i+1) >= '0' && *(s+i+1) <= '9') {

      if (sscanf(s+i+1, "%d%n", &decimal, &expo) > 0)
        ret = 1;
      i += expo;
    }

  } while (*(s+(++i)) != 0 && !ret);

  if (!ret)
    return 0;

  *num = i2fpt(whole) + fpt_div(i2fpt(decimal), i2fpt(_pow(10,expo)));

  if (neg)
    *num = fpt_mul(*num, FPT_MINUS_ONE);

  *br = i;

  return ret;

}

/**
 * Convert the given fixedpt number to a decimal string.
 * The max_dec argument specifies how many decimal digits to the right
 * of the decimal point to generate. If set to -1, the "default" number
 * of decimal digits will be used (2 for 32-bit fixedpt width, 10 for
 * 64-bit fixedpt width); If set to -2, "all" of the digits will
 * be returned, meaning there will be invalid, bogus digits outside the
 * specified precisions.
 */
//static inline ssize_t
ssize_t fpt_str(fpt A, char *str, int max_dec)
{
  int ndec = 0, slen = 0;
  char tmp[12] = {0};
  fptud fr, ip;
  const fptud one = (fptud)1 << FPT_BITS;
  const fptud mask = one - 1;
```

```
  if (max_dec == -1)
#if FPT_BITS == 32
#if FPT_WBITS > 16
    max_dec = 2;
#else
    max_dec = 4;
#endif
#elif FPT_BITS == 64
    max_dec = 10;
#else
#error Invalid width
#endif
  else if (max_dec == -2)
    max_dec = 15;

  if (A < 0) {
    str[slen++] = '-';
    A *= -1;
  }

  ip = fpt2i(A);
  do {
    tmp[ndec++] = '0' + ip % 10;
    ip /= 10;
  } while (ip != 0);

  while (ndec > 0)
    str[slen++] = tmp[--ndec];
  str[slen++] = '.';

  fr = (fpt_fracpart(A) << FPT_WBITS) & mask;
  do {
    fr = (fr & mask) * 10;

    str[slen++] = '0' + (fr >> FPT_BITS) % 10;
    ndec++;
  } while (fr != 0 && ndec < max_dec);

  if (ndec > 1 && str[slen-1] == '0')
    str[slen-1] = '\0'; /* cut off trailing 0 */
  else
    str[slen] = '\0';

  return (ssize_t) slen;
}

/* Converts the given fixedpt number into a string, using a static
 * (non-threadsafe) string buffer */
//static inline char*
char * fpt_cstr(const fpt A, const int max_dec)
{
      static char str[25];

      fpt_str(A, str, max_dec);
      return (str);
}

/* Returns the square root of the given number, or -1 in case of error */
```

```
//static inline fpt
fpt fpt_sqrt(fpt A)
{
  int invert = 0;
  int iter = FPT_FBITS;
  int l, i;

  if (A < 0)
    return (-1);
  if (A == 0 || A == FPT_ONE)
    return (A);
  if (A < FPT_ONE && A > 6) {
    invert = 1;
    A = fpt_div(FPT_ONE, A);
  }
  if (A > FPT_ONE) {
    int s = A;

    iter = 0;
    while (s > 0) {
      s >>= 2;
      iter++;
    }
  }

  /* Newton's iterations */
  l = (A >> 1) + 1;
  for (i = 0; i < iter; i++)
    l = (l + fpt_div(A, l)) >> 1;
  if (invert)
    return (fpt_div(FPT_ONE, l));
  return (l);
}


/* Returns the sine of the given fpt number.
 * Note: the loss of precision is extraordinary! */
//static inline fpt
fpt fpt_sin(fpt fp)
{
  int sign = 1;
  fpt sqr, result;
  const fpt SK[2] = {
    fl2fpt(7.61e-03),
    fl2fpt(1.6605e-01)
  };

  fp %= 2 * FPT_PI;
  if (fp < 0)
    fp = FPT_PI * 2 + fp;
  if ((fp > FPT_HALF_PI) && (fp <= FPT_PI))
    fp = FPT_PI - fp;
  else if ((fp > FPT_PI) && (fp <= (FPT_PI + FPT_HALF_PI))) {
    fp = fp - FPT_PI;
    sign = -1;
  } else if (fp > (FPT_PI + FPT_HALF_PI)) {
    fp = (FPT_PI << 1) - fp;
    sign = -1;
  }
```

```
  sqr = fpt_mul(fp, fp);
  result = SK[0];
  result = fpt_mul(result, sqr);
  result -= SK[1];
  result = fpt_mul(result, sqr);
  result += FPT_ONE;
  result = fpt_mul(result, fp);
  return sign * result;
}


/* Returns the cosine of the given fpt number */
//static inline fpt
fpt fpt_cos(fpt A)
{
  return (fpt_sin(FPT_HALF_PI - A));
}


/* Returns the tangens of the given fpt number */
//static inline fpt
fpt fpt_tan(fpt A)
{
  return fpt_div(fpt_sin(A), fpt_cos(A));
}


/* Returns the value exp(x), i.e. e^x of the given fpt number. */
//static inline fpt
fpt fpt_exp(fpt fp)
{
  fpt xabs, k, z, R, xp;
  const fpt LN2 = fl2fpt(0.69314718055994530942);
  const fpt LN2_INV = fl2fpt(1.4426950408889634074);
  const fpt EXP_P[5] = {
    fl2fpt(1.66666666666666019037e-01),
    fl2fpt(-2.77777777770155933842e-03),
    fl2fpt(6.61375632143793436117e-05),
    fl2fpt(-1.65339022054652515390e-06),
    fl2fpt(4.13813679705723846039e-08),
  };

  if (fp == 0)
    return (FPT_ONE);
  xabs = fpt_abs(fp);
  k = fpt_mul(xabs, LN2_INV);
  k += FPT_ONE_HALF;
  k &= ~FPT_FMASK;
  if (fp < 0)
    k = -k;
  fp -= fpt_mul(k, LN2);
  z = fpt_mul(fp, fp);
  /* Taylor */
  R = FPT_TWO +
      fpt_mul(z, EXP_P[0] + fpt_mul(z, EXP_P[1] +
      fpt_mul(z, EXP_P[2] + fpt_mul(z, EXP_P[3] +
      fpt_mul(z, EXP_P[4])))));
  xp = FPT_ONE + fpt_div(fpt_mul(fp, FPT_TWO), R - fp);
  if (k < 0)
```

```
    k = FPT_ONE >> (-k >> FPT_FBITS);
  else
    k = FPT_ONE << (k >> FPT_FBITS);
  return (fpt_mul(k, xp));
}


/* Returns the natural logarithm of the given fpt number. */
//static inline fpt
fpt fpt_ln(fpt x)
{
  fpt log2, xi;
  fpt f, s, z, w, R;
  const fpt LN2 = fl2fpt(0.69314718055994530942);
  const fpt LG[7] = {
    fl2fpt(6.66666666666735130e-01),
    fl2fpt(3.99999999940941908e-01),
    fl2fpt(2.85714287436623914e-01),
    fl2fpt(2.22221984321497839e-01),
    fl2fpt(1.81835721616180501e-01),
    fl2fpt(1.53138376992093733e-01),
    fl2fpt(1.47981986051165859e-01)
  };

  if (x < 0)
    return (0);
  if (x == 0)
    return 0xffffffff;

  log2 = 0;
  xi = x;
  while (xi > FPT_TWO) {
    xi >>= 1;
    log2++;
  }
  f = xi - FPT_ONE;
  s = fpt_div(f, FPT_TWO + f);
  z = fpt_mul(s, s);
  w = fpt_mul(z, z);
  R = fpt_mul(w, LG[1] + fpt_mul(w, LG[3]
      + fpt_mul(w, LG[5]))) + fpt_mul(z, LG[0]
      + fpt_mul(w, LG[2] + fpt_mul(w, LG[4]
      + fpt_mul(w, LG[6]))));
  return (fpt_mul(LN2, (log2 << FPT_FBITS)) + f
      - fpt_mul(s, f - R));
}


/* Returns the logarithm of the given base of the given fpt number */
// static inline fpt
fpt fpt_log(fpt x, fpt base)
{
  return (fpt_div(fpt_ln(x), fpt_ln(base)));
}


/* Return the power value (n^exp) of the given fpt numbers */
//static inline fpt
fpt fpt_pow(fpt n, fpt exp)
```

```
{
  if (exp == 0)
    return (FPT_ONE);
  if (n < 0)
    return 0;
  return (fpt_exp(fpt_mul(fpt_ln(n), exp)));
}

#endif
```