OBJECT rt_fix **IS PASSIVE**
DESCRIPTION
This module implements fixed point arithmetic for rt-linux. This code is influenced by:

- some code written by Bob Pendleton (can be found somewhere on the web).
- a fixed-point simulation utility written by Seehyun Kim and Prof. Wonjong Sung, VLSI Signal Processing Laboratory, School of Electrical Engineering Seoul National University (→http://mpeg.snu.ac.kr).
- the Fix class implemented in the GNU-C++ Library (→/usr/include/g++).

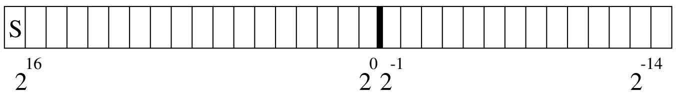
In contrast to the fixed point implementations mentioned above the functionality supplied by this module follows different goals. These goals are:

- applyability in user- and in kernel space,
- implementation of trigonometric functions and
- implementation of other functions contained in the normal math library.

Thus a shared library called `librt_fix.so` and a kernel module called `rt_fix.o` are supplied.

The base type for a fixed point number is called `rt_fix` and contains a 32-bit word used to represent the fixed point number. The length of the integral part and of the fractional part of the fixed point number is determined when a variable of type `rt_fix` is initialized. The internal representation is always kept left aligned and only as many bits as needed are used for the integral part. This aims to keeping as much precision for the fractional part as possible.

The following example show the layout of a fixed point number with 17-bits for the integral part, 14-bits for the fractional part and one sign-bit.



S = Sign Bit

This means that the following numeric limits are given for the integral part:

- High Value = 131071
- Low Value = -131072

The accuracy of the fixed point number is 2^{-14} which is four digits after the decimal point.

The goal of using fixed point arithmetic is to avoid bothering with the FPU in kernel space. This seems reasonable, because using the FPU in kernel space is a constant source of trouble. The drawback is the limited accuracy of and programming overhead caused by fixed point arithmetic.

Doubles used in user space can be converted to fixed point numbers before sending them to rtlinux (and thus to kernel space) and in kernel space calculations should be done using fixed point arithmetic. If in a overflow in a called function occurs the overflowed fixed point number gets saturated. The same action takes place, if an attempt to divide by zero is made.

It is planned to implement the functions shown in the table below (a + in the status column means available and − means not available):

Name	C-equivalent	Description	State
`rt_fix_div()`	/ operator	calculates a / b	+
`rt_fix_mul()`	* operator	calculates a * b	+
`rt_fix_add()`	+ operator	calculates a + b	+
`rt_fix_sub()`	- operator	calculates a - b	+
`rt_fix_toi()`	cast operator	converts a rt_fix variable into a integer number.	+
`rt_fix_init_i()`	none	initialize a rt_fix variable with a integer value.	+
`rt_fix_tod()`	cast operator	converts a rt_fix variable into a double number.	+
`rt_fix_init_d()`	none	initialize a rt_fix variable with a double number.	+
`rt_fix_frac()`	none	extracts the fractional part from a rt_fix variable.	+
`rt_fix_ceil()`	ceil()	extract smallest integral fixed point variable not less than given rt_fix variable.	+
`rt_fix_floor()`	floor()	extract largest integral fixed point variable not greater than given rt_fix variable.	+
`rt_fix_modf()`	modf()	extracts signed integral and fractional part from a rt_fix variable.	+
`rt_fix_abs()`	abs()	computes the absolute value of a rt_fix variable.	+
`rt_fix_sin()`	sin()	sine function.	+
`rt_fix_asin()`	asin()	arc sine function.	+
`rt_fix_cos()`	cos()	cosine function.	+
`rt_fix_acos()`	acos()	arc cosine function.	+
`rt_fix_tan()`	tan()	tangent function.	+
`rt_fix_atan()`	atan()	arc tangent function.	+
`rt_fix_ldexp()`	ldexp()	multiply given rt_fix variable by integral power of 2.	+
`rt_fix_sqrt()`	sqrt()	square root.	+
`rt_fix_pow()`	pow()	x raised to the power of y.	+
`rt_fix_log()`	log()	natural logarithm of a given rt_fix number.	+
`rt_fix_log2()`	none	logarithm to the base of two.	+
`rt_fix_log10()`	log10()	base-10 logarithm of a given rt_fix number.	+
`rt_fix_exp()`	exp()	e to the power of given rt_fix number.	+
`rt_fix_sinh()`	sinh()	hyperbolic sine function.	+
`rt_fix_cosh()`	cosh()	hyperbolic cosine function.	+
`rt_fix_tanh()`	tanh()	hyperbolic tangent function.	+

IMPLEMENTATION CONSTRAINTS

The multiplication function `rt_fix_mul` and the division function `rt_fix_div` are implemented using inline assembly language. The functions `rt_fix_tod` and `rt_fix_tof` are not available in kernel space.

3a ⟨*PROVIDED* 3a⟩≡
⟨*PROLOGUE* 3b⟩
⟨*PROVIDED INCLUDES* 3c⟩
⟨*PROVIDED TYPES* 4b⟩
⟨*PROVIDED CONSTANTS* 4a⟩
⟨*PROVIDED OPERATION SETS* 32a⟩
⟨*PROVIDED OPERATIONS* 5⟩
⟨*EPILOGUE* 32b⟩

3b ⟨*PROLOGUE* 3b⟩≡

```
#ifndef _RT_FIX_H_
#define _RT_FIX_H_
/*
 * Fixed Point Arithmetic Routines
 *
 * 1998 Till Christian Siering
 *
 */
```

3c ⟨*PROVIDED INCLUDES* 3c⟩≡

```
#include <values.h>           /* Imports MAXLONG and MINLONG */

#if defined(__KERNEL__)
#   include <math.h>
#elif
#   include <linux/kernel.h>  /* needed for debian 2.0 libc2 */
#endif /* __KERNEL__ */
```

4a ⟨*PROVIDED CONSTANTS* 4a⟩≡

```
/*
 * Number of Bits used to represent the integral and fractional part
 * of a fixed point number. Bit 32 is used for the sign bit in the
 * internal two's complement representation.
 */
#define RT_FIXWL    31

/*
 * Default integer word length.
 */
#define DEFAULT_IWL     17

/*
 * Size of a string to print the string representation of a fixed point number
 * into.
 */
#define RT_FIX_STR_REP_SIZE 30

/*
 * Overflow bit.
 */
#define RT_FIX_OV        1

/*
 * Underflow bit.
 */
#define RT_FIX_UV        2

/*
 * Undefined result bit.
 */
#define RT_FIX_UNDEFINED 4

extern const rt_fix rt_fix_0;   /* = 0         */
extern const rt_fix rt_fix_1;   /* = 1         */
extern const rt_fix rt_fix_min; /* = -2^31     */
extern const rt_fix rt_fix_max; /* = 2^31 - 1  */
extern const rt_fix rt_fix_pi;  /* = 3.141...  */
```

4b ⟨*PROVIDED TYPES* 4b⟩≡

```
typedef struct _rt_fix {
    long _value;              /* value of this object                        */
    unsigned short _iwl;      /* integer word-length => bits for the mantissa  */
    unsigned short _errorset; /* set for recording exceptions                  */
} rt_fix;
```

5 ⟨*PROVIDED OPERATIONS* 5⟩≡

```
/*
 * clear the error set of a fixed point number.
 * InOut:
 *  rt_fix *fix pointer to a fixed point number.
 * Post:
 *  All errors in the error set of fix are cleared.
 * Out:
 *   0  if all errors could be cleared
 *  -1  on error
 */
extern __inline__ short rt_fix_erremptyset(rt_fix *fix)
{
    if(!fix) {
        return -1;
    }
    fix->_errorset = 0;
    return 0;

} /* End of rt_fix_erremptyset */

/*
 * add a given error to the error set of a fixed point number.
 * InOut:
 *  rt_fix *fix pointer to a fixed point number.
 * In:
 *  unsigned short errnum   the error to be added
 * Post:
 *  the error is added to the error set of fix.
 * Out:
 *   0  if the error could be added
 *  -1  on error
 */
extern __inline__ short rt_fix_erraddset(rt_fix *fix, unsigned short errnum)
{
    if(!fix) {
        return -1;
    }
    switch(errnum) {
        case RT_FIX_OV:
        case RT_FIX_UV:
        case RT_FIX_UNDEFINED:
            fix->_errorset |= errnum;
            return 0;
        default:
            return -1;
    }
} /* End of rt_fix_erraddset */

/*
 * merge the error sets of two fixed point numbers.
 * InOut:
 *  rt_fix *target pointer to a fixed point number.
 * In:
```

```
 *  rt_fix source  a fixed point number.
 * Post:
 *  the error set of source is added to the error set of target.
 * Out:
 *   0  if the error sets could be merged
 *  -1  on error
 */
extern __inline__ short rt_fix_errmergeset(rt_fix *target, rt_fix source)
{
    if(!target) {
        return -1;
    }
    target->_errorset |= source._errorset;
    return 0;

} /* End of rt_fix_errmergeset */

/*
 * delete a given error from the error set of a fixed point number.
 * InOut:
 *  rt_fix *fix pointer to a fixed point number.
 * In:
 *  unsigned short errnum   the error to be deleted
 * Post:
 *  the error is deleted from the error set of fix.
 * Out:
 *   0  if the error could be deleted
 *  -1  on error
 */
extern __inline__ short rt_fix_errdelset(rt_fix *fix, unsigned short errnum)
{
    if(!fix) {
        return -1;
    }
    switch(errnum) {
        case RT_FIX_OV:
        case RT_FIX_UV:
        case RT_FIX_UNDEFINED:
            fix->_errorset &= ~errnum;
            return 0;
        default:
            return -1;
    }
} /* End of rt_fix_errdelset */

/*
 * set all possible errors in the error set of a fixed point number.
 * InOut:
 *  rt_fix *fix pointer to a fixed point number.
 * Post:
 *  all possible errors are set in the error set of fix.
 * Out:
 *   0  if all errors could be set
 *  -1  on error
```

```
 */
extern __inline__ short rt_fix_errfillset(rt_fix *fix)
{
    short result = 0;

    if(!fix) {
        return -1;
    }
    if(rt_fix_erraddset(fix, RT_FIX_OV) == -1) {
        result = -1;
    }
    if(rt_fix_erraddset(fix, RT_FIX_UV) == -1) {
        result = -1;
    }
    if(rt_fix_erraddset(fix, RT_FIX_UNDEFINED) == -1) {
        result = -1;
    }
    return result;

} /* End of rt_fix_errfillset */

/*
 * test, if a fixed point number is error is faulty.
 * In:
 *  rt_fix fix  a fixed point number.
 * Out:
 *   1  if fix is faulty
 *   0  otherwise
 */
extern __inline__ short rt_fix_errtstset(rt_fix fix)
{
    return (fix._errorset != 0);

} /* End of rt_fix_errtstset */

/*
 * test, if a error is set in the error set of a fixed point number.
 * In:
 *  rt_fix fix  a fixed point number
 *  short errnum    the error to be tested
 * Out:
 *   1  if errnum is a member of the error set of fix
 *   0  otherwise
 */
extern __inline__ short rt_fix_errisset(rt_fix fix, unsigned short errnum)
{
    switch(errnum) {
        case RT_FIX_OV:
        case RT_FIX_UV:
        case RT_FIX_UNDEFINED:
            if(fix._errorset & errnum) {
                return 1;
            }
        default:
```

```
            return 0;
    }
} /* End of rt_fix_errisset */

/*
 * get the sign of a fixed point number.
 * In:
 *  rt_fix arg  the fixed point number to get the
 *              sign from.
 * Out:
 *  1   if arg is < 0
 *  0   else
 */
extern __inline__ short rt_fix_sign(rt_fix arg)
{
    arg._value &= MINLONG;
    return arg._value;

} /* End of rt_fix_sign */

/*
 * Computes the absolute value of x.
 * Out:
 *  The absolute value of x.
 */
extern __inline__ rt_fix rt_fix_abs(rt_fix fix)
{
    if(fix._value < 0) {
        fix._value = -fix._value;
        return fix;
    }
    return fix;

} /* End of rt_fix_abs */

/*
 * negate a fixed point number (multiply it by -1).
 * In:
 *  rt_fix arg  the fixed point number to be negated.
 * Out:
 *  A fixed point number, which is the result of arg * -1.
 */
extern __inline__ rt_fix rt_fix_neg(rt_fix arg)
{
    arg._value = -arg._value;
    return arg;
} /* End of rt_fix_neg */

/*
 * initialize a fixed point number with default values.
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *              is manipulated
 * Post:
```

```
 *  The fixed point number fix is set to 0 and its
 *  integer word length is set to DEFAULT_IWL.
 */
extern __inline__ short rt_fix_init(rt_fix *fix)
{
    if(!fix) {
        return 1;
    }
    fix->_iwl = DEFAULT_IWL;
    fix->_value = 0;
    rt_fix_erremptyset(fix);
    return 0;

} /* End of rt_fix_init */

/*
 * initialize a fixed point number with a given integer.
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *              is initialized
 * In:
 *  int i   the integer used for initialization
 * Out:
 *  0   on success
 *  1   on error
 * Post:
 *  The fixed point number fix is initialized with the
 *  given integer.
 */
extern __inline__ short rt_fix_init_i(rt_fix *fix, int i)
{
    unsigned short iwl = 1;
    int p = 1;
    int tmp_i = i;

    if(!fix) {
        return 1;
    }
    /*
     * find out how many bits are needed to represent i.
     */
    if(tmp_i < 0) {
        tmp_i = -i;
    }
    for(iwl = 1; iwl <= RT_FIXWL; iwl++) {
        if(tmp_i <= p) {
            break;
        }
        p <<= 1;
    }
    fix->_iwl = iwl;
    fix->_value = (long)(i << (RT_FIXWL - fix->_iwl));
    rt_fix_erremptyset(fix);
    return 0;
```

```
}  /* End of rt_fix_init_i */

/*
 * initialize a fixed point number with a given long.
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *              is initialized
 * In:
 *  int l   the long used for initialization
 * Out:
 *  0   on success
 *  1   on error
 * Post:
 *  The fixed point number fix is initialized with the
 *  given long.
 */
extern __inline__ short rt_fix_init_l(rt_fix *fix, long l)
{
    unsigned short iwl = 1;
    int p = 1;

    if(!fix) {
        return 1;
    }
    /*
     * find out how many bits are needed to represent l.
     */
    for(iwl = 1; iwl <= RT_FIXWL; iwl++) {
        if(l <= p) break;
        p <<= 1;
    }
    fix->_iwl = iwl;
    fix->_value = (long)(l << (RT_FIXWL - fix->_iwl));
    rt_fix_erremptyset(fix);
    return 0;

} /* End of rt_fix_init_l */

/*
 * initialize with given 32-Bit-Mask and given integer word length.
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *              is initialized
 * In:
 *  long bit_mask           the bit_mask used for initialization
 *  unsigned short  iwlen   the integer word length used for initialization
 * Out:
 *  0   on success
 *  1   on error
 * Post:
 *  The fixed point number fix is initialized with the
 *  given bit mask and the given integer word length.
 */
```

```
extern __inline__ short rt_fix_init_mask(rt_fix *fix,
                                         long bit_mask,
                                         unsigned short iwlen)
{
    if(!fix) {
        return 1;
    }
    fix->_iwl = iwlen;
    fix->_value = bit_mask;
    rt_fix_erremptyset(fix);
    return 1;

} /* End of rt_fix_init_mask */

#ifndef __KERNEL__
/*
 * initialize a fixed pointer number with a given double.
 * InOut:
 *  rt_fix *fix point to the fixed point number which
 *              is initialized
 * In:
 *  double d   the double used for initialization
 * Out:
 *  0   on success
 *  1   on error
 * Post:
 *  The fixed point number fix is initialized with the
 *  given double.
 */
extern __inline__ short rt_fix_init_d(rt_fix *fix, double d)
{
    unsigned int iwl = 1;
    int p = 1;

    if(!fix) {
        return 1;
    }
    /*
     * find out how many bits are needed to represent d.
     */
    for(iwl = 1; iwl <= RT_FIXWL; iwl++) {
        if(fabs(d) <= p) break;
        p <<= 1;
    }
    if(iwl > RT_FIXWL) {
        if(d >= 0.) {
            *fix = rt_fix_max;
        } else {
            *fix = rt_fix_min;
        }
        rt_fix_erraddset(fix, RT_FIX_OV);
    } else {
        fix->_iwl = iwl;
        fix->_value = (long)(d * (1 << (RT_FIXWL - fix->_iwl)));
```

```
        rt_fix_erremptyset(fix);
    }
    return 0;

} /* End of rt_fix_init_d */
#endif

/*
 * transform a fixed point number to long.
 * In:
 *  rt_fix fix  the fixed point number to be
 *              transformed
 * Out:
 *  The result of the transformation.
 */
extern __inline__ long rt_fix_tol(rt_fix fix)
{
    return((long)(fix._value >> (RT_FIXWL - fix._iwl)));

} /* End of rt_fix_tol */

/*
 * transform a fixed point number to a int.
 * In:
 *  rt_fix fix  the fixed point number to be
 *              transformed
 * Out:
 *  The result of the transformation.
 */
extern __inline__ int rt_fix_toi(rt_fix fix)
{
    return((int)(fix._value >> (RT_FIXWL - fix._iwl)));

} /* End of rt_fix_toi */

#ifndef __KERNEL__
/*
 * transform a fixed point number to a double.
 * In:
 *  rt_fix fix  the fixed point number to be
 *              transformed
 * Out:
 *  The result of the transformation.
 */
extern __inline__ double rt_fix_tod(rt_fix fix)
{
    double result;

    result = (double)fix._value;
    return(result / (1 << (RT_FIXWL - fix._iwl)));

} /* End of rt_fix_tod */
#endif
```

```
/*
 * shift a fixed point number b bits left.
 * In:
 *  rt_fix arg  the fixed point number to be shifted
 *  int b       the number of bits arg is shifted.
 * Pre:
 *  The sum of the integer word length of arg and b is
 *  smaller or equal RT_FIXWL.
 * Out:
 *  A fixed point number representing arg * 2^b.
 */
extern __inline__ rt_fix rt_fix_shl(rt_fix arg, int b)
{
    arg._value <<= b;
    return arg;

} /* End of rt_fix_shl */

/*
 * shift a fixed point number b bits right.
 * In:
 *  rt_fix arg  the fixed point number to be shifted
 *  int b       the number of bits arg is shifted.
 * Out:
 *  A fixed point number representing arg / 2^b.
 */
extern __inline__ rt_fix rt_fix_shr(rt_fix arg, int b)
{
    arg._value >>= b;
    return arg;

} /* End of rt_fix_shr */

#ifdef NEEDED
/*
 * Transform the internal representation of a fixed point number
 * into the internal representation with minimal integer word length.
 * InOut:
 *  rt_fix fix  fixed point number whose integer word length is to
 *              be minimized.
 * Post:
 *  The integer word length of the fixed point number fix is minimized.
 *  The smallest integer word length is one.
 */
extern __inline__ void rt_fix_min_iwl(rt_fix *fix)
{
    unsigned long mask = 1073741824; /* = 2^30 */

    if(fix) {
        /*
         * integer word length of fix is minimal
         */
        if(fix->_iwl < 2) {
            return;
```

```
        }
        /*
         * test if fix is negative
         */
        if(fix->_value < 0) {
            fix->_value = ~fix->_value;
            while(!(fix->_value & mask) && fix->_iwl > 1) {
                fix->_value <<= 1;
                fix->_iwl--;
            }
            fix->_value = ~fix->_value;
        } else {
            while(!(fix->_value & mask) && fix->_iwl > 1) {
                fix->_value <<= 1;
                fix->_iwl--;
            }
        }
    }
} /* End of rt_fix_min_iwl */
#endif

/*
 * Transform the internal representation of a fixed point number
 * into the internal representation with minimal integer word length.
 * InOut:
 *  rt_fix fix  fixed point number whose integer word length is to
 *              be minimized.
 * Post:
 *  The integer word length of the fixed point number fix is minimized.
 *  The smallest integer word length is one.
 */
extern __inline__ void rt_fix_min_iwl(rt_fix *fix)
{
    unsigned long mask = 1073741824;  /* = 2^30 */
    short sign = 0;

    if(fix) {
        if(fix->_iwl < 2) {
            return;
        }
        sign = rt_fix_sign(*fix);
        if(sign) {
            *fix = rt_fix_neg(*fix);
            while(!(fix->_value & mask) && fix->_iwl > 1) {
                fix->_value <<= 1;
                fix->_iwl--;
            }
            *fix = rt_fix_neg(*fix);
        } else {
            while(!(fix->_value & mask) && fix->_iwl > 1) {
                fix->_value <<= 1;
                fix->_iwl--;
            }
        }
```

```
    }
} /* End of rt_fix_min_iwl */

/*
 * Add two fixed point numbers.
 * In:
 *  rt_fix x  the summand
 *  rt_fix y  the fixed point number added to x
 * Out:
 *  A fixed point number representing the result of x + y.
 *  If the result overflows rt_fix_max is returned.
 */
extern __inline__ rt_fix rt_fix_add(rt_fix x, rt_fix y)
{
    unsigned short iwl = (x._iwl > y._iwl) ? x._iwl + 1 : y._iwl + 1;

    /*
     * let the result appear in x.
     */
    rt_fix_errmergeset(&x, y);
    if(iwl > RT_FIXWL) {
        x = rt_fix_max;
        rt_fix_errmergeset(&x, y);
        rt_fix_erraddset(&x, RT_FIX_OV);
        return x;
    }
    if(x._iwl > y._iwl) {
        x._value = (x._value >> 1) + (y._value >> (x._iwl - y._iwl + 1));
    } else if(x._iwl < y._iwl) {
        x._value = (x._value >> (y._iwl - x._iwl + 1)) + (y._value >> 1);
    } else {
        x._value = (x._value >> 1) + (y._value >> 1);
    }
    x._iwl = iwl;
    rt_fix_min_iwl(&x);
    return x;

} /* End of rt_fix_add */

/*
 * Subtract to fixed point numbers.
 * In:
 *  rt_fix x  the minuend
 *  rt_fix y  the subtrahend
 * Out:
 *  A fixed point number representing the result of x - y.
 *  If the result overflows rt_fix_max is returned.
 */
extern __inline__ rt_fix rt_fix_sub(rt_fix x, rt_fix y)
{
    unsigned short iwl = (x._iwl > y._iwl) ? x._iwl + 1 : y._iwl + 1;

    /*
     * let the result appear in x.
```

```
     */
    rt_fix_errmergeset(&x, y);
    if(iwl > RT_FIXWL) {
        x = rt_fix_max;
        rt_fix_errmergeset(&x, y);
        rt_fix_erraddset(&x, RT_FIX_OV);
        return x;
    }
    if(x._iwl > y._iwl) {
        x._value = (x._value >> 1) - (y._value >> (x._iwl - y._iwl + 1));
    } else if(x._iwl < y._iwl) {
        x._value = (x._value >> (y._iwl - x._iwl + 1)) - (y._value >> 1);
    } else {
        x._value = (x._value >> 1) - (y._value >> 1);
    }
    x._iwl = iwl;
    rt_fix_min_iwl(&x);
    return x;

} /* End of rt_fix_sub */

/*
 * Multiply two fixed point numbers.
 * In:
 *  rt_fix x  the multiplicand
 *  rt_fix y  the multiplicator
 * Out:
 *  A fixed point number representing the result of x * y.
 *  If the result overflows rt_fix_max is returned.
 */
extern __inline__ rt_fix rt_fix_mul(rt_fix x, rt_fix y)
{
    unsigned short iwl = x._iwl + y._iwl;

    /*
     * let the result appear in x.
     */
    rt_fix_errmergeset(&x, y);
    if(iwl > RT_FIXWL) {
        if(rt_fix_sign(x) ^ rt_fix_sign(y)) {
            x = rt_fix_min;
        } else {
            x = rt_fix_max;
        }
        rt_fix_errmergeset(&x, y);
        rt_fix_erraddset(&x, RT_FIX_OV);
        return x;
    }
    if(y._iwl < iwl) {
        y._value >>= x._iwl;
    }
    if(x._iwl < iwl) {
        x._value >>= y._iwl;
    }
```

```
    x._iwl = iwl;
    __asm__("imull %%ebx\n\t"             /* Multiply a * b */
            "shrdl %%cl, %%edx, %%eax\n\t"
            : "=a" (x._value)
            : "a"  (x._value), "b" (y._value), "c" (RT_FIXWL - x._iwl)
            : "edx");
    rt_fix_min_iwl(&x);
    return x;

} /* End of rt_fix_mul */

/*
 * Divide two fixed point numbers.
 * In:
 *  rt_fix x  the divident
 *  rt_fix y  the divisor
 * Out:
 *  A fixed point number representing the result of x / y.
 *  If the attempt to divide by zero is made rt_fix_max
 *  is returned.
 */
extern __inline__ rt_fix rt_fix_div(rt_fix x, rt_fix y)
{
    unsigned short iwl = x._iwl;
    short sign;
    long tmp;

    sign = rt_fix_sign(x) ^ rt_fix_sign(y);
    x = rt_fix_abs(x); y = rt_fix_abs(y);

    /*
     * let the result appear in x.
     */
    rt_fix_errmergeset(&x, y);
    if(y._value == 0) {
        /*
         * division by zero
         */
        if(sign) {
            x = rt_fix_min;
        } else {
            x = rt_fix_max;
        }
        rt_fix_errmergeset(&x, y);
        rt_fix_erraddset(&x, RT_FIX_UNDEFINED);
        return x;
    }
    if(x._iwl > y._iwl) {
        /*
         * |x| > |y|
         */
        y._value >>= (x._iwl - y._iwl);
        iwl = x._iwl;
        if(y._value == 0) {
```

```
        if(sign) {
            x = rt_fix_min;
        } else {
            x = rt_fix_max;
        }
        rt_fix_errmergeset(&x, y);
        rt_fix_erraddset(&x, RT_FIX_UNDEFINED);
        return x;
    }
} else if(x._iwl < y._iwl) {
    /*
     * |x| < |y|
     */
    x._value >>= (y._iwl - x._iwl);
    iwl = y._iwl;
    if(x._value == 0) {
        return rt_fix_0;
    }
}
if(rt_fix_toi(y) == 0 || rt_fix_toi(y) == -1) {
    tmp = y._value;
    /*
     * |y| < 1
     */
    tmp <<= y._iwl;
    while(!(tmp & (1 << RT_FIXWL))) {
        tmp <<= 1;
        iwl++;
    }
    iwl = iwl + x._iwl;
    if(iwl > RT_FIXWL) {
        if(sign) {
            x = rt_fix_min;
        } else {
            x = rt_fix_max;
        }
        rt_fix_errmergeset(&x, y);
        rt_fix_erraddset(&x, RT_FIX_OV);
        return x;
    }
}
x._iwl = iwl;
__asm__("xorl  %%edx, %%edx\n\t"       /* Clear edx                             */
        "shldl %%cl, %%eax, %%edx\n\t" /* Shift edx left and fill in from eax */
        "sall  %%cl, %%eax\n\t"        /* Shift eax upward                      */
        "idivl %%ebx\n\t"              /* Divide (edx:eax) / ebx                */
        : "=a" (x._value)
        : "a"  (x._value),
          "b"  (y._value),
          "c"  ((unsigned short) (RT_FIXWL - x._iwl))
        : "edx");
rt_fix_min_iwl(&x);
if(sign) {
    return rt_fix_neg(x);
```

```
    } else {
        return x;
    }
} /* End of rt_fix_div */

/*
 * Test if two fixed point numbers are equal.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x and y are equal
 *  0   otherwise
 */
extern __inline__ short rt_fix_eq(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
        x._value >>= (y._iwl - x._iwl);
    }
    return(x._value == y._value);

} /* End of rt_fix_eq */

/*
 * Test if two fixed point numbers are not equal.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x and y are not equal
 *  0   otherwise
 */
extern __inline__ short rt_fix_neq(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
        x._value >>= (y._iwl - x._iwl);
    }
    return(x._value != y._value);

} /* End of rt_fix_neq */

/*
 * Test if two fixed point numbers are greater-equal.
 * In:
```

```
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x is greater or equal y
 *  0   otherwise
 */
extern __inline__ short rt_fix_geq(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
        x._value >>= (y._iwl - x._iwl);
    }
    if(x._value >= y._value) {
        return 1;
    } else {
        return 0;
    }
} /* End of rt_fix_geq */

/*
 * Test if two fixed point numbers are smaller-equal.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x is smaller or equal y
 *  0   otherwise
 */
extern __inline__ short rt_fix_leq(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
        x._value >>= (y._iwl - x._iwl);
    }
    if(x._value <= y._value) {
        return 1;
    } else {
        return 0;
    }
} /* End of rt_fix_leq */

/*
 * Test if one fixed point numbers is greater than another.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
```

```
 * Out:
 *  1   if x is greater y
 *  0   otherwise
 */
extern __inline__ short rt_fix_gtr(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
        x._value >>= (y._iwl - x._iwl);
    }
    if(x._value > y._value) {
        return 1;
    } else {
        return 0;
    }
} /* End of rt_fix_gtr */

/*
 * Test if one fixed point number is smaller than another.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x is smaller y
 *  0   otherwise
 */
extern __inline__ short rt_fix_lss(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
        x._value >>= (y._iwl - x._iwl);
    }
    if(x._value < y._value) {
        return 1;
    } else {
        return 0;
    }
} /* End of rt_fix_lss */

/************************
 * builtin math functions
 ***********************/

/*
 * Multiply a fixed point number by integral power of 2.
 * In:
```

```
 *  rt_fix fix  fixed point number to be mulitplied
 *  int pow2    exponent of pow(2, pow2)
 * Out:
 *  On success the result of the multiplication.
 *  On error rt_fix_max.
 */
#define rt_fix_ldexp(fix, pow2) (rt_fix_shl(arg, b))

/*
 * Get the fractional part of a fixed point number.
 * In:
 *  rt_fix fix  fixed point number to extract the
 *              fractional part from.
 * Out:
 *  A fixed point number which contains the fractional
 *  part of fix.
 */
extern __inline__ rt_fix rt_fix_frac(rt_fix fix)
{
    unsigned int p = 1;
    long value = fix._value;
    unsigned int i;

    fix._value = 0;
    if(value < 0) {
        value = -value;
        for(i = 0; i < (RT_FIXWL - fix._iwl); i++) {
            if((p & value) > 0) {
                fix._value += p;
            }
            p <<= 1;
        }
        fix._value = -fix._value;
    } else {
        for(i = 0; i < (RT_FIXWL - fix._iwl); i++) {
            if((p & value) > 0) {
                fix._value += p;
            }
            p <<= 1;
        }
    }
    return fix;

} /* End of frac */

/*
 * Calculate the largest integral value of a fixed point number
 * not greater than the value of the fixed point number itself.
 * In:
 *  rt_fix fix  the fixed point number to calculate the largest
 *              integral value from.
 * Out:
 *  A fixed point number which represents the the largest integral
 *  value not greater than the value of fix.
```

```
 */
extern __inline__ rt_fix rt_fix_floor(rt_fix fix)
{
    rt_fix fraction = rt_fix_frac(fix);

    if(0 != fraction._value) {
        if(fix._value < 0) {
            fix._value = fix._value - fraction._value - (1 << (RT_FIXWL - fix._iwl));
            return fix;
        } else {
            fix._value = fix._value - fraction._value;
            return fix;
        }
    }
    return fix;

} /* End of rt_fix_floor */

/*
 * Calculate the smallest integral value not less than the value of
 * a given fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the largest
 *              integral value from.
 * Out:
 *  A fixed point number which represents the smallest integral
 *  value not less than the value of the called object.
 */
extern __inline__ rt_fix rt_fix_ceil(rt_fix fix)
{
    rt_fix fraction = rt_fix_frac(fix);

    if(0 != fraction._value) {
        if(fix._value < 0) {
            fix._value = fix._value - fraction._value;
            return fix;
        } else {
            fix._value = fix._value + (1 << (RT_FIXWL - fix._iwl)) - fraction._value;
            return fix;
        }
    }
    return fix;

} /* End of rt_fix_ceil */

/*
 * Extract signed integral and fractional part of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to extract the parts from
 * InOut:
 *  rt_fix *iptr    pointer to a fixed point number in which the integral
 *                  part will be stored.
 * Out:
 *  A fixed point number which represents the fractional part of fix.
```

```
 */
extern __inline__ rt_fix rt_fix_modf(rt_fix fix, rt_fix *iptr)
{
    rt_fix fraction = rt_fix_frac(fix);

    if(iptr) {
        iptr->_value = fix._value - fraction._value;
        iptr->_iwl = fix._iwl;
        iptr->_errorset = fix._errorset;
    }
    return fraction;

} /* End of rt_fix_modf */

/*
 * Round a fixed point number to nearest integral value.
 * In:
 *  rt_fix fix  the fixed point number which is rounded
 * Out:
 *  A fixed point number which represents the rounded value
 *  of fix.
 */
extern __inline__ rt_fix rt_fix_round(rt_fix fix)
{
    rt_fix fraction = rt_fix_abs(rt_fix_frac(fix));

    fraction._value >>= (RT_FIXWL - fraction._iwl) - 1;
    if(fix._value >= 0) {
        if(fraction._value > 0) {
            return rt_fix_ceil(fix);
        }  else {
            return rt_fix_floor(fix);
        }
    } else {
        if(fraction._value > 0) {
            return rt_fix_floor(fix);
        }  else {
            return rt_fix_ceil(fix);
        }
    }
} /* End of rt_fix_round */

/*
 * Calculate the sine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the sine from
 * Pre:
 *  |fix| <= 1024, to get a result with a precision of
 *  approximately 1E-4.
 * Out:
 *  A fixed point number which represents the approximated
 *  sine of fix.
 */
```

```
extern rt_fix rt_fix_sin(rt_fix fix);

/*
 * Calculate the arc sine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the arc sine from
 * Pre:
 *  fix e [-1, 1].
 * Out:
 *  A fixed point number which represents the approximated
 *  arc sine of fix.
 */
extern rt_fix rt_fix_asin(rt_fix fix);

/*
 * Calculate the cosine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              cosine from
 * Pre:
 *  |fix| <= 1024, to get a result with a precision of
 *  approximately 1E-4.
 * Out:
 *  A fixed point number which represents the approximated
 *  cosine of fix.
 */
extern rt_fix rt_fix_cos(rt_fix fix);

/*
 * Calculate the arc cosine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              arc cosine from
 * Pre:
 *  fix e [-1, 1].
 * Out:
 *  A fixed point number which represents the approximated
 *  arc cosine of fix.
 */
extern rt_fix rt_fix_acos(rt_fix fix);

/*
 * Calculate the tangent of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              arc tangent from
 * Pre:
 *  |fix| <= 1024, to get a result with a precision of
 *  approximately 1E-4.
 * Out:
 *  The approximated tangent of the rt_fix number fix.
 */
extern rt_fix rt_fix_tan(rt_fix fix);
```

```
/*
 * Calculate the arc tangent of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              arc tangent from
 * Out:
 *  A fixed point number which represents the approximated
 *  arc tangent of fix.
 */
extern rt_fix rt_fix_atan(rt_fix fix);

/*
 * Calculate the square root of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the square root from
 * Out:
 *  - A fixed point number which represents the approximated
 *    square root of fix.
 *  - If fix is negative the square root of the absolute value
 *    of fix is calculated.
 */
extern rt_fix rt_fix_sqrt(rt_fix fix);

/*
 * Calculate fix raised to the power of e.
 * In:
 *  rt_fix fix  the fixed point number to be raised to the
 *              power of e.
 * Out:
 *  A fixed point number which represents the result of
 *  the calculation.
 */
extern rt_fix rt_fix_exp(rt_fix fix);

/*
 * Calculate the logarithm to the base of two.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the logarithm to the base of two from.
 * Out:
 *  - A fixed point number which represents the result of
 *    the calculation.
 *  - If fix is negative the logarithm to the base of two of
 *    the absolute value of fix is calculated.
 *  - If fix is zero rt_fix_min is returned.
 */
extern rt_fix rt_fix_log2(rt_fix x);

/*
 * Calculate the natural logarithm of fix.
 * In:
 *  rt_fix fix  the fixed point number to calculate
```

```
 *               the natural logarithm from.
 * Out:
 *  - A fixed point number which represents the result of
 *    the calculation.
 *  - If fix is negative the natural logarithm of the
 *    absolute value of fix is calculated.
 *  - If fix is zero rt_fix_min is returned.
 */
extern __inline__ rt_fix rt_fix_log(rt_fix fix)
{
    rt_fix ln_2 = {1488522236, 0, 0}; /* = 0.69314718 */

    return rt_fix_mul(ln_2, rt_fix_log2(fix));

} /* End of rt_fix_log */

/*
 * Calculate the decimal logarithm of fix.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the decimal logarithm from.
 * Out:
 *  - A fixed point number which represents the result of
 *    the calculation.
 *  - If fix is negative the logarithm to the base of ten of
 *    the absolute value of fix is calculated.
 *  - If fix is zero rt_fix_min is returned.
 */
extern __inline__ rt_fix rt_fix_log10(rt_fix fix)
{
    rt_fix log_2 = {646456993, 0, 0}; /* = 0.301029995 */

    return rt_fix_mul(log_2, rt_fix_log2(fix));

} /* End of rt_fix_log10 */

/*
 * Calculate x raised to the power of y.
 * In:
 *  rt_fix fix  the fixed point number to be raised
 *              to the power of y.
 *  rt_fix y    the exponent.
 * Pre:
 *  If y is not an integral value then x must be positive.
 * Out:
 *  - A fixed point number which represents the result of
 *    the calculation.
 *  - If fix is negative and y is not an integral value then
 *    the absolute value of fix is used to calculate the
 *    result and RT_FIX_UNDEFINED is set in the error set
 *    of the result.
 */
extern __inline__ rt_fix rt_fix_pow(rt_fix fix, rt_fix y)
{
```

```
    if(rt_fix_lss(fix, rt_fix_0) && rt_fix_neq(rt_fix_frac(y), rt_fix_0)) {
        fix = rt_fix_abs(fix);
        fix = rt_fix_exp(rt_fix_mul(y, rt_fix_log(fix)));
        rt_fix_erraddset(&fix, RT_FIX_UNDEFINED);
        return fix;
    }
    return rt_fix_exp(rt_fix_mul(y, rt_fix_log(fix)));

} /* End of rt_fix_pow */

/*
 * Calculate the hyperbolic sine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              hyperbolic sine from.
 * Out:
 *  A fixed point number which represents the result of
 *  the calculation.
 */
extern __inline__ rt_fix rt_fix_sinh(rt_fix fix)
{
    rt_fix one_half = {1073741824, 0, 0}; /* = 0.5 */

    return rt_fix_mul(rt_fix_sub(rt_fix_exp(fix),
                                 rt_fix_exp(rt_fix_neg(fix))),
                      one_half);

} /* End of rt_fix_sinh */

/*
 * Calculate the hyperbolic cosine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              hyperbolic cosine from.
 * Out:
 *  A fixed point number which represents the result of
 *  the calculation.
 */
extern __inline__ rt_fix rt_fix_cosh(rt_fix fix)
{
    rt_fix one_half = {1073741824, 0, 0}; /* = 0.5 */

    return rt_fix_mul(rt_fix_add(rt_fix_exp(fix),
                                 rt_fix_exp(rt_fix_neg(fix))),
                      one_half);

} /* End of rt_fix_cosh */

/*
 * Calculate the hyperbolic tangent of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              hyperbolic tangent from.
 * Out:
```

```
 *  A fixed point number which represents the result of
 *  the calculation.
 */
extern __inline__ rt_fix rt_fix_tanh(rt_fix fix)
{
    return rt_fix_div(rt_fix_sub(rt_fix_exp(fix),
                                 rt_fix_exp(rt_fix_neg(fix))),
                      rt_fix_add(rt_fix_exp(fix),
                                 rt_fix_exp(rt_fix_neg(fix))));

} /* End of rt_fix_tanh */

#ifdef RT_FIXMISC
/*
 * assign a fixed point number to another fixed point number without
 * changing the integer word lenght of the fixed point number
 * to which is assigned.
 * InOut:
 *  rt_fix *left    pointer to fixed point number which is assigned to
 * In:
 *  rt_fix right    fixed point number which is assigned to left
 * Pre:
 *  The largest value the fixed point number left can be assigned to
 *  is equal or greater than the fixed point number right.
 * Post:
 *  The fixed point number left has the same value as the fixed
 *  point number right.  The integer word length of the fixed
 *  point number left is not changed.
 * Out:
 *  0   on success
 *  1   on error
 */
extern short rt_fix_assfix(rt_fix *left, rt_fix right);

/*
 * assign a integer number to a fixed point number without
 * changing the integer word lenght of the fixed point number
 * to which is assigned.
 * InOut:
 *  rt_fix *left    pointer to fixed point number which is
 *                  assigned to
 * In:
 *  int i           integer number which is assigned to left
 * Pre:
 *  The largest value the fixed point number left can be assigned to
 *  is equal or greater than i.
 * Post:
 *  The fixed point number left has the value i. The integer word
 *  length of the fixed point number left is not changed.
 * Out:
 *  0   on success
 *  1   on error (overflow)
 */
extern short rt_fix_assi(rt_fix *left, int i);
```

```
/*
 * assign a long number to a fixed point number without changing
 * the integer word length of the fixed point number to which
 * is assigned.
 * InOut:
 *  rt_fix *left    pointer to fixed point number which is
 *                  assigned to
 * In:
 *  long l          long number which is assigned to left
 * Pre:
 *  The largest value the fixed point number left can be assigned to
 *  is equal or greater than l.
 * Post:
 *  The fixed point number left has the value l. The integer word
 *  length of the fixed point number left is not changed.
 * Out:
 *  0   on success
 *  1   on error (overflow)
 */
extern short rt_fix_assl(rt_fix *left, long l);

#ifndef __KERNEL__
/*
 * assign a double number to a fixed point number without
 * changing the integer word length of the fixed point number
 * to which is assigned.
 * InOut:
 *  rt_fix *left    pointer to fixed point number which is
 *                  assigned to
 * In:
 *  double d        double number which is assigned to left
 * Post:
 *  The number left has the value d. The fraction word length
 *  of the left is not changed.
 * Out:
 *  0   on success
 *  1   on error (overflow)
 */
extern short rt_fix_assd(rt_fix *left, double d);
#endif /* __KERNEL__ */

/*
 * Set the integer word length of a fixed point number.
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *              is manipulated
 * In:
 *  unsigned short iwlen  the new integer word length
 * Post:
 *  The integer word length of the fixed point number is
 *  set to iwlen.
 * Out:
 *  0   on success
```

```
 *  1   on error
 */
extern __inline__ short rt_fix_set_iwl(rt_fix *fix, unsigned short iwlen)
{
    if(!fix) {
        return 1;
    }
    fix->_iwl = iwlen;
    return 0;

} /* End of rt_fix_set_iwl */

/*
 * query integer word length of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to query the
 *              integer word length from
 * Out:
 *  The integer word length of fix.
 */
extern __inline__ unsigned short rt_fix_get_iwl(rt_fix fix)
{
    return fix._iwl;

} /* End of rt_fix_get_iwl */

/*
 * query internal representation
 * In:
 *  rt_fix fix  the fixed point number to query the
 *              internal representation from
 * Out:
 *  The internal representation of fix.
 */
extern __inline__ long rt_fix_get_rep(rt_fix fix)
{
    return fix._value;

} /* End of rt_fix_get_rep */

/*
 * convert a fixed point number into its string representation.
 * The string representation has the following format:
 *  internal_value  integer-word-length fraction-word-length
 *     = long           = int                 = int
 * In:
 *  rt_fix fix  the fixed point number to print
 *  char *str_rep   the string to print the representation
 *                  into.
 * Pre:
 *  str_rep is not NULL and its length is equal or greater than
 *  RT_FIX_STR_REP_SIZE.
 * Post:
 *  str_rep contains the string representation of the fixed point
```

```
 *  number fix.
 * Out:
 *  0   on success
 *  1   on error
 */
extern short rt_fix_toa(rt_fix fix, char *str_rep);

/*
 * convert a given string representation of a fixed point number into
 * a fixed point number.
 * In:
 *  const char *str_rep  the string representation of a fixed point
 *                        number.
 *  rt_fix *result      the fixed point number to store the result of
 *                      the conversion into.
 * Pre:
 *  str_rep and result are not NULL and str_rep contains a valid string
 *  representation of a fixed point number.
 * Post:
 *  The fixed point number *result contains the fixed point number
 *  described by string str_rep.
 * Out:
 *  0   on success
 *  1   on error
 */
extern short rt_fix_atofix(rt_fix *result, const char *string_rep);

#endif /* RT_FIXMISC */
```

32a ⟨*PROVIDED OPERATION SETS* 32a⟩≡

None.

32b ⟨*EPILOGUE* 32b⟩≡

```
#endif /* _RT_FIX_H_ */
```

OBJECT CONTROL STRUCTURE

None.

CONSTRAINED OPERATIONS

None.

REQUIRED INTERFACE

None.

32c ⟨*INTERNA* 32c⟩≡

⟨*INTERNAL PROLOGUE* 33a⟩
⟨*INTERNAL INCLUDES* 33b⟩
⟨*INTERNAL CONSTANTS* 33c⟩
⟨*INTERNAL TYPES* 33d⟩
⟨*INTERNAL GLOBALS* 33e⟩
⟨*INTERNAL OPERATION SETS* 33f⟩
⟨*IMPLEMENTATION OF INTERNAL OPERATIONS* 35⟩
⟨*IMPLEMENTATION OF PROVIDED CONSTANTS* 34⟩
⟨*IMPLEMENTATION OF PROVIDED OPERATIONS* 37⟩

33a ⟨*INTERNAL PROLOGUE* 33a⟩≡
```
/*
 * Fixed Point Arithmetic Routines
 *
 * 1998 Till Christian Siering
 */
#ifdef __KERNEL__
#define MODULE
#endif /* __KERNEL__ */
```

33b ⟨*INTERNAL INCLUDES* 33b⟩≡
```
#ifdef __KERNEL__
#include <linux/module.h>  /* Imports Module Definitions           */
#include <linux/kernel.h>  /* Imports some often used functions    */
#include <linux/version.h> /* Imports version number               */
#include <linux/config.h>  /* Imports some often used functions    */
#include <linux/stddef.h>  /* Imports NULL                         */
#include <linux/string.h>  /* Imports memset                       */
#else
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#endif /* __KERNEL__ */
#include <rt_fix.h>        /* Imports declarations and definitions *
                            * for fixed point arithmetic           */
```

33c ⟨*INTERNAL CONSTANTS* 33c⟩≡
```
/*
 * Largest value for a rt_fix number.
 */
#define RT_FIXMAX   MAXLONG

/*
 * Smallest value for a rt_fix number.
 */
#define RT_FIXMIN   MINLONG
```

33d ⟨*INTERNAL TYPES* 33d⟩≡

None.

33e ⟨*INTERNAL GLOBALS* 33e⟩≡

None.

33f ⟨*INTERNAL OPERATION SETS* 33f⟩≡

None.

34 ⟨*IMPLEMENTATION OF PROVIDED CONSTANTS* 34⟩≡

```
const rt_fix rt_fix_0 = {
    0,                /* value of this variable */
    0,                /* integer word-length    */
    0                 /* no errors              */
};

const rt_fix rt_fix_1 = {
    1073741824,       /* value of this variable */
    1,                /* integer word-length    */
    0                 /* no errors              */
};

const rt_fix rt_fix_pi = {
    1686629713,       /* value of this variable */
    2,                /* integer word-length    */
    0                 /* no errors              */
};

const rt_fix rt_fix_min = {
    RT_FIXMIN,         /* value of this variable */
    RT_FIXWL,          /* integer word-length    */
    0                  /* no errors              */
};

const rt_fix rt_fix_max = {
    RT_FIXMAX,         /* value of this variable */
    RT_FIXWL,          /* integer word-length    */
    0                  /* no errors              */
};
```

35 ⟨*IMPLEMENTATION OF INTERNAL OPERATIONS* 35⟩≡

```
/*
 * approximate sine and cosine for fix e [0, 2 * PI].
 * In:
 *  rt_fix fix  the number to approximate the sine or
 *              cosine from.
 *  int cos     = 1 => cosine is approximated.
 * Pre:
 *  fix e [0, 2 * PI]
 * Out:
 *  The result of the approximation.
 */
__inline__ static rt_fix
rt_fix_sincos(rt_fix fix, int cos)
{
    int octant;
    rt_fix x_4, x_2, int_part;
    rt_fix four_pi = {1367130551, 1, 0}; /* = 4 / PI */

    fix = rt_fix_mul(four_pi, fix);
    fix = rt_fix_modf(fix, &int_part);
    octant = rt_fix_toi(int_part);
    if(cos) {
        octant = octant + 2;
    }
    x_2 = rt_fix_mul(fix, fix);
    x_4 = rt_fix_mul(x_2, x_2);
    switch(octant) {
        case 1:
        case 3:
        case 5:
        case 7:
            fix = rt_fix_sub(rt_fix_1, fix);
            break;
    }
    x_2 = rt_fix_mul(fix, fix);
    x_4 = rt_fix_mul(x_2, x_2);
    switch(octant) {
        case 0:
        case 3:
        case 4:
        case 7: {
            rt_fix p[] = {
                { 1772298122, 6, 0},       /* = 52.81860  */
                {-1246829006, 3, 0},       /* = -4.64480  */
                {  186303870, 0, 0}        /* = 0.0867545 */
            };
            rt_fix q = {1128280023, 7, 0}; /* 67.25073 */

            fix = rt_fix_mul(fix,
                    rt_fix_div(
                      rt_fix_add(
                        rt_fix_add(p[0], rt_fix_mul(p[1], x_2)),
                        rt_fix_mul(p[2], x_4)),
```

```
                    rt_fix_add(q, x_2)));
            }
            break;
        case 1:
        case 2:
        case 5:
        case 6: {
            rt_fix p[] = {
                { 1600119930, 6, 0},        /* = 47.68729 */
                {-1839856615, 4, 0},        /* = -13.7080 */
                {  961690422, 0, 0}         /* = 0.447822 */
            };

            fix = rt_fix_div(
                    rt_fix_add(
                      rt_fix_add(p[0], rt_fix_mul(p[1], x_2)),
                      rt_fix_mul(p[2], x_4)),
                    rt_fix_add(p[0], x_2));
            }
            break;
    }
    switch(octant) {
        case 4:
        case 5:
        case 6:
        case 7:
            fix = rt_fix_neg(fix);
            break;
    }
    /*
     * merge all errors which appeared in intermediate
     * calculations into the result.
     */
    rt_fix_errmergeset(&fix, x_2);
    rt_fix_errmergeset(&fix, x_4);
    return fix;

} /* End of rt_fix_sincos */
```

37 ⟨*IMPLEMENTATION OF PROVIDED OPERATIONS* 37⟩≡

```
/*
 * clear the error set of a fixed point number.
 * InOut:
 *  rt_fix *fix pointer to a fixed point number.
 * Post:
 *  All errors in the error set of fix are cleared.
 * Out:
 *   0  if all errors could be cleared
 *  -1  on error
 */
short
rt_fix_erremptyset(rt_fix *fix)
{
    if(!fix) {
        return -1;
    }
    fix->_errorset = 0;
    return 0;

} /* End of rt_fix_erremptyset */

/*
 * add a given error to the error set of a fixed point number.
 * InOut:
 *  rt_fix *fix pointer to a fixed point number.
 * In:
 *  unsigned short errnum   the error to be added
 * Post:
 *  the error is added to the error set of fix.
 * Out:
 *   0  if the error could be added
 *  -1  on error
 */
short
rt_fix_erraddset(rt_fix *fix, unsigned short errnum)
{
    if(!fix) {
        return -1;
    }
    switch(errnum) {
        case RT_FIX_OV:
        case RT_FIX_UV:
        case RT_FIX_UNDEFINED:
            fix->_errorset |= errnum;
            return 0;
        default:
            return -1;
    }
} /* End of rt_fix_erraddset */

/*
 * merge the error sets of two fixed point numbers.
 * InOut:
```

```
 *  rt_fix *target pointer to a fixed point number.
 * In:
 *  rt_fix source  a fixed point number.
 * Post:
 *  the error set of source is added to the error set of target.
 * Out:
 *   0  if the error sets could be merged
 *  -1  on error
 */
short
rt_fix_errmergeset(rt_fix *target, rt_fix source)
{
    if(!target) {
        return -1;
    }
    target->_errorset |= source._errorset;
    return 0;

} /* End of rt_fix_errmergeset */

/*
 * delete a given error from the error set of a fixed point number.
 * InOut:
 *  rt_fix *fix pointer to a fixed point number.
 * In:
 *  unsigned short errnum   the error to be deleted
 * Post:
 *  the error is deleted from the error set of fix.
 * Out:
 *   0  if the error could be deleted
 *  -1  on error
 */
short
rt_fix_errdelset(rt_fix *fix, unsigned short errnum)
{
    if(!fix) {
        return -1;
    }
    switch(errnum) {
        case RT_FIX_OV:
        case RT_FIX_UV:
        case RT_FIX_UNDEFINED:
            fix->_errorset &= ~errnum;
            return 0;
        default:
            return -1;
    }
} /* End of rt_fix_errdelset */

/*
 * set all possible errors in the error set of a fixed point number.
 * InOut:
 *  rt_fix *fix pointer to a fixed point number.
 * Post:
```

```
 *  all possible errors are set in the error set of fix.
 * Out:
 *   0  if all errors could be set
 *  -1  on error
 */
short
rt_fix_errfillset(rt_fix *fix)
{
    short result = 0;

    if(!fix) {
        return -1;
    }
    if(rt_fix_erraddset(fix, RT_FIX_OV) == -1) {
        result = -1;
    }
    if(rt_fix_erraddset(fix, RT_FIX_UV) == -1) {
        result = -1;
    }
    if(rt_fix_erraddset(fix, RT_FIX_UNDEFINED) == -1) {
        result = -1;
    }
    return result;

} /* End of rt_fix_errfillset */

/*
 * test, if a fixed point number is error is faulty.
 * In:
 *  rt_fix fix  a fixed point number.
 * Out:
 *   1  if fix is faulty
 *   0  otherwise
 */
short
rt_fix_errtstset(rt_fix fix)
{
    return (fix._errorset != 0);

} /* End of rt_fix_errtstset */

/*
 * test, if a error is set in the error set of a fixed point number.
 * In:
 *  rt_fix fix  a fixed point number
 *  unsigned short errnum   the error to be tested
 * Out:
 *   1  if errnum is a member of the error set of fix
 *   0  otherwise
 */
short
rt_fix_errisset(rt_fix fix, unsigned short errnum)
{
    switch(errnum) {
```

```
        case RT_FIX_OV:
        case RT_FIX_UV:
        case RT_FIX_UNDEFINED:
            if(fix._errorset & errnum) {
                return 1;
            }
        default:
            return 0;
    }
} /* End of rt_fix_errisset */

/*
 * get the sign of a fixed point number.
 * In:
 *  rt_fix arg  the fixed point number to get the
 *              sign from.
 * Out:
 *  1   if arg is < 0
 *  0   else
 */
short
rt_fix_sign(rt_fix arg)
{
    if(arg._value &= MINLONG) {
        return 1;
    } else {
        return 0;
    }
} /* End of rt_fix_sign */

/*
 * negate a fixed point number (multiply it by -1).
 * In:
 *  rt_fix arg  the fixed point number to be negated.
 * Out:
 *  A fixed point number, which is the result of arg * -1.
 */
rt_fix
rt_fix_neg(rt_fix arg)
{
    arg._value = -arg._value;
    return arg;
} /* End of rt_fix_neg */

#ifdef NEEDED
/*
 * Transform the internal representation of a fixed point number
 * into the internal representation with minimal integer word length.
 * InOut:
 *  rt_fix fix  fixed point number whose integer word length is to
 *              be minimized.
 * Post:
 *  The integer word length of the fixed point number fix is minimized.
 *  The smallest integer word length is one.
```

```
 */
void
rt_fix_min_iwl(rt_fix *fix)
{
    unsigned long mask = 1073741824; /* = 2^30 */

    if(fix) {
        /*
         * integer word length of fix is minimal
         */
        if(fix->_iwl < 2) {
            return;
        }
        /*
         * test if fix is negative
         */
        if(fix->_value < 0) {
            fix->_value = ~fix->_value;
            while(!(fix->_value & mask) && fix->_iwl > 1) {
                fix->_value <<= 1;
                fix->_iwl--;
            }
            fix->_value = ~fix->_value;
        } else {
            while(!(fix->_value & mask) && fix->_iwl > 1) {
                fix->_value <<= 1;
                fix->_iwl--;
            }
        }
    }
} /* End of rt_fix_min_iwl */
#endif

/*
 * Transform the internal representation of a fixed point number
 * into the internal representation with minimal integer word length.
 * InOut:
 *  rt_fix fix  fixed point number whose integer word length is to
 *              be minimized.
 * Post:
 *  The integer word length of the fixed point number fix is minimized.
 *  The smallest integer word length is one.
 */
void
rt_fix_min_iwl(rt_fix *fix)
{
    unsigned long mask = 1073741824;  /* = 2^30 */
    short sign = 0;

    if(fix) {
        if(fix->_iwl < 2) {
            return;
        }
        sign = rt_fix_sign(*fix);
```

```
        if(sign) {
            *fix = rt_fix_neg(*fix);
            while(!(fix->_value & mask) && fix->_iwl > 1) {
                fix->_value <<= 1;
                fix->_iwl--;
            }
            *fix = rt_fix_neg(*fix);
        } else {
            while(!(fix->_value & mask) && fix->_iwl > 1) {
                fix->_value <<= 1;
                fix->_iwl--;
            }
        }
    }
} /* End of rt_fix_min_iwl */

/*
 * initialize a fixed point number with default values.
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *              is initialized
 * Post:
 *  The fixed point number fix is set to 0 and its
 *  integer word length is set to DEFAULT_IWL.
 */
short
rt_fix_init(rt_fix *fix)
{
    if(!fix) {
        return 1;
    }
    fix->_iwl = DEFAULT_IWL;
    fix->_value = 0;
    rt_fix_erremptyset(fix);
    return 0;

} /* End of rt_fix_init */

/*
 * initialize a fixed point number with a given integer.
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *              is initialized
 * In:
 *  int i   the integer used for initialization
 * Out:
 *  0   on success
 *  1   on error
 * Post:
 *  The fixed point number fix is initialized with the
 *  given integer.
 */
short
rt_fix_init_i(rt_fix *fix, int i)
```

```
{
    unsigned short iwl = 1;
    int p = 1;
    int tmp_i = i;

    if(!fix) {
        return 1;
    }
    /*
     * find out how many bits are needed to represent i.
     */
    if(tmp_i < 0) {
        tmp_i = -i;
    }
    for(iwl = 1; iwl <= RT_FIXWL; iwl++) {
        if(tmp_i <= p) {
            break;
        }
        p <<= 1;
    }
    fix->_iwl = iwl;
    fix->_value = (long)(i << (RT_FIXWL - fix->_iwl));
    rt_fix_erremptyset(fix);
    return 0;

}  /* End of rt_fix_init_i */

/*
 * initialize a fixed point number with a given long.
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *             is initialized
 * In:
 *  long l  the long used for initialization
 * Out:
 *  0   on success
 *  1   on error
 * Post:
 *  The fixed point number fix is initialized with the
 *  given long.
 */
short
rt_fix_init_l(rt_fix *fix, long l)
{
    unsigned short iwl = 1;
    int p = 1;

    if(!fix) {
        return 1;
    }
    /*
     * find out how many bits are needed to represent l.
     */
    for(iwl = 1; iwl <= RT_FIXWL; iwl++) {
```

```
        if(l <= p) break;
        p <<= 1;
    }
    fix->_iwl = iwl;
    fix->_value = (long)(l << (RT_FIXWL - fix->_iwl));
    rt_fix_erremptyset(fix);
    return 0;

} /* End of rt_fix_init_l */

/*
 * initialize with given 32-Bit-Mask and given integer word length.
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *              is initialized
 * In:
 *  long bit_mask           the bit_mask used for initialization
 *  unsigned short  iwlen   the integer word length used for initialization
 * Out:
 *  0   on success
 *  1   on error
 * Post:
 *  The fixed point number fix is initialized with the
 *  given bit mask and the given integer word length.
 */
short
rt_fix_init_mask(rt_fix *fix, long bit_mask, unsigned short iwlen)
{
    if(!fix) {
        return 1;
    }
    fix->_iwl = iwlen;
    fix->_value = bit_mask;
    rt_fix_erremptyset(fix);
    return 1;

} /* End of rt_fix_init_mask */

#ifndef __KERNEL__
/*
 * construct from a double value
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *              is initialized
 * In:
 *  double d    the double used for initialization
 * Out:
 *  0   on success
 *  1   on error
 * Post:
 *  The fixed point number fix is initialized with the
 *  given double.
 */
short
```

```
rt_fix_init_d(rt_fix *fix, double d)
{
    unsigned int iwl = 1;
    int p = 1;

    if(!fix) {
        return 1;
    }
    /*
     * find out how many bits are needed to represent d.
     */
    for(iwl = 1; iwl <= RT_FIXWL; iwl++) {
        if(fabs(d) <= p) break;
        p <<= 1;
    }
    if(iwl > RT_FIXWL) {
        if(d >= 0.) {
            *fix = rt_fix_max;
        } else {
            *fix = rt_fix_min;
        }
        rt_fix_erraddset(fix, RT_FIX_OV);
    } else {
        fix->_iwl = iwl;
        fix->_value = (long)(d * (1 << (RT_FIXWL - fix->_iwl)));
        rt_fix_erremptyset(fix);
    }
    return 0;

} /* End of rt_fix_init_d */
#endif /* __KERNEL__ */

/*
 * transform a fixed point number to a long.
 * In:
 *  rt_fix fix  the fixed point number to be
 *              transformed
 * Out:
 *  The result of the transformation.
 */
long
rt_fix_tol(rt_fix fix)
{
    return((long)(fix._value >> (RT_FIXWL - fix._iwl)));

} /* End of rt_fix_tol */

/*
 * transform a fixed point number to a int.
 * In:
 *  rt_fix fix  the fixed point number to be
 *              transformed
 * Out:
 *  The result of the transformation.
```

```
 */
int
rt_fix_toi(rt_fix fix)
{
    return((int)(fix._value >> (RT_FIXWL - fix._iwl)));

} /* End of rt_fix_toi */

#ifndef __KERNEL__
/*
 * transform a fixed point number to a double.
 * In:
 *  rt_fix fix  the fixed point number to be
 *              transformed
 * Out:
 *  The result of the transformation.
 */
double
rt_fix_tod(rt_fix fix)
{
    double result;

    result = (double)fix._value;
    return(result / (1 << (RT_FIXWL - fix._iwl)));

} /* End of rt_fix_tod */
#endif /* __KERNEL__ */

/*
 * shift a fixed point number b bits left.
 * In:
 *  rt_fix arg  the fixed point number to be shifted
 *  int b       the number of bits arg is shifted.
 * Pre:
 *  The sum of the integer word length of arg and b is
 *  smaller or equal RT_FIXWL.
 * Out:
 *  A fixed point number representing arg * 2^b.
 */
rt_fix
rt_fix_shl(rt_fix arg, int b)
{
    arg._value <<= b;
    return arg;

} /* End of rt_fix_shl */

/*
 * shift a fixed point number b bits right.
 * In:
 *  rt_fix arg  the fixed point number to be shifted
 *  int b       the number of bits arg is shifted.
 * Out:
 *  A fixed point number representing arg / 2^b.
```

```
 */
rt_fix
rt_fix_shr(rt_fix arg, int b)
{
    arg._value >>= b;
    return arg;

} /* End of rt_fix_shr */

/*
 * Add two fixed point numbers.
 * In:
 *  rt_fix x  the summand
 *  rt_fix y  the fixed point number added to x
 * Out:
 *  A fixed point number representing the result of x + y.
 *  If the result overflows rt_fix_max is returned.
 */
rt_fix
rt_fix_add(rt_fix x, rt_fix y)
{
    unsigned short iwl = (x._iwl > y._iwl) ? x._iwl + 1 : y._iwl + 1;

    /*
     * let the result appear in x.
     */
    rt_fix_errmergeset(&x, y);
    if(iwl > RT_FIXWL) {
        x = rt_fix_max;
        rt_fix_errmergeset(&x, y);
        rt_fix_erraddset(&x, RT_FIX_OV);
        return x;
    }
    if(x._iwl > y._iwl) {
        x._value = (x._value >> 1) + (y._value >> (x._iwl - y._iwl + 1));
    } else if(x._iwl < y._iwl) {
        x._value = (x._value >> (y._iwl - x._iwl + 1)) + (y._value >> 1);
    } else {
        x._value = (x._value >> 1) + (y._value >> 1);
    }
    x._iwl = iwl;
    rt_fix_min_iwl(&x);
    return x;

} /* End of rt_fix_add */

/*
 * Subtract to fixed point numbers.
 * In:
 *  rt_fix x  the minuend
 *  rt_fix y  the subtrahend
 * Out:
 *  A fixed point number representing the result of x - y.
 *  If the result overflows rt_fix_max is returned.
```

```
 */
rt_fix
rt_fix_sub(rt_fix x, rt_fix y)
{
    unsigned short iwl = (x._iwl > y._iwl) ? x._iwl + 1 : y._iwl + 1;

    /*
     * let the result appear in x.
     */
    rt_fix_errmergeset(&x, y);
    if(iwl > RT_FIXWL) {
        x = rt_fix_max;
        rt_fix_errmergeset(&x, y);
        rt_fix_erraddset(&x, RT_FIX_OV);
        return x;
    }
    if(x._iwl > y._iwl) {
        x._value = (x._value >> 1) - (y._value >> (x._iwl - y._iwl + 1));
    } else if(x._iwl < y._iwl) {
        x._value = (x._value >> (y._iwl - x._iwl + 1)) - (y._value >> 1);
    } else {
        x._value = (x._value >> 1) - (y._value >> 1);
    }
    x._iwl = iwl;
    rt_fix_min_iwl(&x);
    return x;

} /* End of rt_fix_sub */

/*
 * Multiply two fixed point numbers.
 * In:
 *  rt_fix x  the multiplicand
 *  rt_fix y  the multiplicator
 * Out:
 *  A fixed point number representing the result of x * y.
 *  If the result overflows rt_fix_max is returned.
 */
rt_fix
rt_fix_mul(rt_fix x, rt_fix y)
{
    unsigned short iwl = x._iwl + y._iwl;

    /*
     * let the result appear in x.
     */
    rt_fix_errmergeset(&x, y);
    if(iwl > RT_FIXWL) {
        if(rt_fix_sign(x) ^ rt_fix_sign(y)) {
            x = rt_fix_min;
        } else {
            x = rt_fix_max;
        }
        rt_fix_errmergeset(&x, y);
```

```
        rt_fix_erraddset(&x, RT_FIX_OV);
        return x;
    }
    if(y._iwl < iwl) {
        y._value >>= x._iwl;
    }
    if(x._iwl < iwl) {
        x._value >>= y._iwl;
    }
    x._iwl = iwl;
    __asm__("imull %%ebx\n\t"              /* Multiply a * b */
            "shrdl %%cl, %%edx, %%eax\n\t"
            : "=a" (x._value)
            : "a"  (x._value), "b" (y._value), "c" (RT_FIXWL - x._iwl)
            : "edx");
    rt_fix_min_iwl(&x);
    return x;

} /* End of rt_fix_mul */

/*
 * Divide two fixed point numbers.
 * In:
 *  rt_fix x  the divident
 *  rt_fix y  the divisor
 * Out:
 *  A fixed point number representing the result of x / y.
 *  If the attempt to divide by zero is made rt_fix_max
 *  is returned.
 */
rt_fix
rt_fix_div(rt_fix x, rt_fix y)
{
    unsigned short iwl = x._iwl;
    short sign;
    long tmp;

    sign = rt_fix_sign(x) ^ rt_fix_sign(y);
    x = rt_fix_abs(x); y = rt_fix_abs(y);

    /*
     * let the result appear in x.
     */
    rt_fix_errmergeset(&x, y);
    if(y._value == 0) {
        /*
         * division by zero
         */
        if(sign) {
            x = rt_fix_min;
        } else {
            x = rt_fix_max;
        }
        rt_fix_errmergeset(&x, y);
```

```
    rt_fix_erraddset(&x, RT_FIX_UNDEFINED);
    return x;
}
if(x._iwl > y._iwl) {
    /*
     * |x| > |y|
     */
    y._value >>= (x._iwl - y._iwl);
    iwl = x._iwl;
    if(y._value == 0) {
        if(sign) {
            x = rt_fix_min;
        } else {
            x = rt_fix_max;
        }
        rt_fix_errmergeset(&x, y);
        rt_fix_erraddset(&x, RT_FIX_UNDEFINED);
        return x;
    }
} else if(x._iwl < y._iwl) {
    /*
     * |x| < |y|
     */
    x._value >>= (y._iwl - x._iwl);
    iwl = y._iwl;
    if(x._value == 0) {
        return rt_fix_0;
    }
}
if(rt_fix_toi(y) == 0 || rt_fix_toi(y) == -1) {
    tmp = y._value;
    /*
     * |y| < 1
     */
    tmp <<= y._iwl;
    while(!(tmp & (1 << RT_FIXWL))) {
        tmp <<= 1;
        iwl++;
    }
    iwl = iwl + x._iwl;
    if(iwl > RT_FIXWL) {
        if(sign) {
            x = rt_fix_min;
        } else {
            x = rt_fix_max;
        }
        rt_fix_errmergeset(&x, y);
        rt_fix_erraddset(&x, RT_FIX_OV);
        return x;
    }
}
x._iwl = iwl;
__asm__("xorl  %%edx, %%edx\n\t"       /* Clear edx                          */
        "shldl %%cl, %%eax, %%edx\n\t" /* Shift edx left and fill in from eax */
```

```
            "sall  %%cl, %%eax\n\t"        /* Shift eax upward                    */
            "idivl %%ebx\n\t"              /* Divide (edx:eax) / ebx              */
            : "=a" (x._value)
            : "a"  (x._value),
              "b"  (y._value),
              "c"  ((unsigned short) (RT_FIXWL - x._iwl))
            : "edx");
    rt_fix_min_iwl(&x);
    if(sign) {
        return rt_fix_neg(x);
    } else {
        return x;
    }
} /* End of rt_fix_div */

/*
 * Test if two fixed point numbers are equal.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x and y are equal
 *  0   otherwise
 */
short
rt_fix_eq(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
        x._value >>= (y._iwl - x._iwl);
    }
    return(x._value == y._value);

} /* End of rt_fix_eq */

/*
 * Test if two fixed point numbers are not equal.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x and y are not equal
 *  0   otherwise
 */
short
rt_fix_neq(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
```

```
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
        x._value >>= (y._iwl - x._iwl);
    }
    return(x._value != y._value);

} /* End of rt_fix_neq */

/*
 * Test if two fixed point numbers are greater-equal.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x is greater or equal y
 *  0   otherwise
 */
short
rt_fix_geq(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
        x._value >>= (y._iwl - x._iwl);
    }
    if(x._value >= y._value) {
        return 1;
    } else {
        return 0;
    }
} /* End of rt_fix_geq */

/*
 * Test if two fixed point numbers are smaller-equal.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x is smaller or equal y
 *  0   otherwise
 */
short
rt_fix_leq(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
```

```
        x._value >>= (y._iwl - x._iwl);
    }
    if(x._value <= y._value) {
        return 1;
    } else {
        return 0;
    }
} /* End of rt_fix_leq */

/*
 * Test if one fixed point numbers is greater than another.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x is greater y
 *  0   otherwise
 */
short
rt_fix_gtr(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
        x._value >>= (y._iwl - x._iwl);
    }
    if(x._value > y._value) {
        return 1;
    } else {
        return 0;
    }
} /* End of rt_fix_gtr */

/*
 * Test if one fixed point number is smaller than another.
 * In:
 *  rt_fix x a fixed point number
 *  rt_fix y a fixed point number
 * Out:
 *  1   if x is smaller y
 *  0   otherwise
 */
short
rt_fix_lss(rt_fix x, rt_fix y)
{
    /*
     * lvalue and rvalue must have the same iwl.
     */
    if(x._iwl > y._iwl) {
        y._value >>= (x._iwl - y._iwl);
    } else if(x._iwl < y._iwl) {
```

```
        x._value >>= (y._iwl - x._iwl);
    }
    if(x._value < y._value) {
        return 1;
    } else {
        return 0;
    }
} /* End of rt_fix_lss */

/*
 * Get the fractional part of a fixed point number.
 * In:
 *  rt_fix fix  fixed point number to extract the
 *              fractional part from.
 * Out:
 *  A fixed point number which contains the fractional
 *  part of fix.
 */
rt_fix
rt_fix_frac(rt_fix fix)
{
    unsigned int p = 1;
    long value = fix._value;
    unsigned int i;

    fix._value = 0;
    if(value < 0) {
        value = -value;
        for(i = 0; i < (RT_FIXWL - fix._iwl); i++) {
            if((p & value) > 0) {
                fix._value += p;
            }
            p <<= 1;
        }
        fix._value = -fix._value;
    } else {
        for(i = 0; i < (RT_FIXWL - fix._iwl); i++) {
            if((p & value) > 0) {
                fix._value += p;
            }
            p <<= 1;
        }
    }
    return fix;

} /* End of frac */

/*
 * Calculate the largest integral value of a fixed point number
 * not greater than the value of the fixed point number itself.
 * In:
 *  rt_fix fix  the fixed point number to calculate the largest
 *              integral value from.
 * Out:
```

```
 *  A fixed point number which represents the the largest integral
 *  value not greater than the value of fix.
 */
rt_fix
rt_fix_floor(rt_fix fix)
{
    rt_fix fraction = rt_fix_frac(fix);

    if(0 != fraction._value) {
        if(fix._value < 0) {
            fix._value = fix._value - fraction._value - (1 << (RT_FIXWL - fix._iwl));
            return fix;
        } else {
            fix._value = fix._value - fraction._value;
            return fix;
        }
    }
    return fix;

} /* End of rt_fix_floor */

/*
 * Calculate the smallest integral value not less than the value of
 * a given fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the largest
 *              integral value from.
 * Out:
 *  A fixed point number which represents the smallest integral
 *  value not less than the value of the called object.
 */
rt_fix
rt_fix_ceil(rt_fix fix)
{
    rt_fix fraction = rt_fix_frac(fix);

    if(0 != fraction._value) {
        if(fix._value < 0) {
            fix._value = fix._value - fraction._value;
            return fix;
        } else {
            fix._value = fix._value + (1 << (RT_FIXWL - fix._iwl)) - fraction._value;
            return fix;
        }
    }
    return fix;

} /* End of rt_fix_ceil */

/*
 * Calculate the absolute value of fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              absolute value from.
```

```
 * Out:
 *  A fixed point number which represents the absolute value
 *  of fix.
 */
rt_fix
rt_fix_abs(rt_fix fix)
{
    if(fix._value < 0) {
        fix._value = -fix._value;
        return fix;
    }
    return fix;

} /* End of rt_fix_abs */

/*
 * Extract signed integral and fractional part of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to extract the parts from
 * InOut:
 *  rt_fix *iptr    pointer to a fixed point number in which the integral
 *                  part will be stored.
 * Out:
 *  A fixed point number which represents the fractional part of fix.
 */
rt_fix
rt_fix_modf(rt_fix fix, rt_fix *iptr)
{
    rt_fix fraction = rt_fix_frac(fix);

    if(iptr) {
        iptr->_value = fix._value - fraction._value;
        iptr->_iwl = fix._iwl;
        iptr->_errorset = fix._errorset;
    }
    return fraction;

} /* End of rt_fix_modf */

/*
 * Round a fixed point number to nearest integral value.
 * In:
 *  rt_fix fix  the fixed point number which is rounded
 * Out:
 *  A fixed point number which represents the rounded value
 *  of fix.
 */
rt_fix
rt_fix_round(rt_fix fix)
{
    rt_fix fraction = rt_fix_abs(rt_fix_frac(fix));

    fraction._value >>= (RT_FIXWL - fraction._iwl) - 1;
    if(fix._value >= 0) {
```

```
        if(fraction._value > 0) {
            return rt_fix_ceil(fix);
        }  else {
            return rt_fix_floor(fix);
        }
    } else {
        if(fraction._value > 0) {
            return rt_fix_floor(fix);
        }  else {
            return rt_fix_ceil(fix);
        }
    }
} /* End of rt_fix_round */

/*
 * Calculate the square root of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the square root from
 * Out:
 *  - A fixed point number which represents the approximated
 *    square root of fix.
 *  - If fix is negative the square root of the absolute value
 *    of fix is calculated.
 */
rt_fix
rt_fix_sqrt(rt_fix fix)
{
    long root;
    long next;
    rt_fix result;

    if(rt_fix_sign(fix)) {
        fix = rt_fix_abs(fix);
        rt_fix_erraddset(&fix, RT_FIX_UNDEFINED);
    }
    if(fix._value < 1) {
        fix = rt_fix_0;
        rt_fix_erraddset(&fix, RT_FIX_UV);
        return fix;
    }
    /*
     * the fraction word length of fix must be even in order
     * to be able to calculate the fraction word length of
     * the result (result._fwl = fix._fwl / 2).
     */
    if((RT_FIXWL - fix._iwl) & 1) {
        fix._value >>= 1;
        fix._iwl++;
    }
    next = (fix._value >> 2);
    do
    {
        root = next;
```

```
        /*
         * division should work in kernel space,
         * because the operands are of type long.
         */
        next = (next + fix._value / next) >> 1;
    } while (root != next);
    rt_fix_init_mask(&result, root, (RT_FIXWL + fix._iwl) >> 1);
    rt_fix_min_iwl(&result);
    return result;

} /* End of rt_fix */

/*
 * Calculate the sine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the sine from
 * Pre:
 *  |fix| <= 1024, to get a result with a precision of
 *  approximately 1E-4.
 * Out:
 *  A fixed point number which represents the approximated
 *  sine of fix.
 */
rt_fix
rt_fix_sin(rt_fix fix)
{
    rt_fix pi   = {1686629713, 2, 0}; /* PI     */
    rt_fix pi_2 = {1686629713, 3, 0}; /* PI * 2 */
    short sign  = rt_fix_sign(fix);

    fix = rt_fix_abs(fix);
    if(rt_fix_gtr(fix, pi_2)) {
        fix = rt_fix_div(fix, pi);
        if(rt_fix_tol(fix) & 0x00000001) {
            fix = rt_fix_add(rt_fix_mul(rt_fix_frac(fix), pi), pi);
        } else {
            fix = rt_fix_mul(rt_fix_frac(fix), pi);
        }
    }
    if(sign) {
        return rt_fix_neg(rt_fix_sincos(fix, 0));
    }
    return rt_fix_sincos(fix, 0);

} /* End of rt_fix_sin */

/*
 * Calculate the arc sine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the arc sine from
 * Pre:
 *  fix e [-1, 1].
```

```
 * Out:
 *  A fixed point number which represents the approximated
 *  arc sine of fix.
 */
rt_fix
rt_fix_asin(rt_fix fix)
{
    rt_fix pi_2 = {1686629713, 1, 0}; /* = 1.570796327 */
    rt_fix x_2 = fix;

    if(rt_fix_gtr(rt_fix_abs(fix), rt_fix_1)) {
        fix = rt_fix_max;
        rt_fix_erraddset(&fix, RT_FIX_UNDEFINED);
        return fix;
    }
    if(rt_fix_eq(rt_fix_abs(fix), rt_fix_1)) {
        if(rt_fix_lss(fix, rt_fix_0)) {
            return rt_fix_neg(pi_2);
        } else {
            return pi_2;
        }
    }
    x_2 = rt_fix_mul(x_2, x_2);
    return rt_fix_atan(rt_fix_div(fix, rt_fix_sqrt(rt_fix_sub(rt_fix_1, x_2))));

} /* End of rt_fix_asin */

/*
 * Calculate the cosine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              cosine from
 * Pre:
 *  |fix| <= 1024, to get a result with a precision of
 *  approximately 1E-4.
 * Out:
 *  A fixed point number which represents the approximated
 *  cosine of fix.
 */
rt_fix
rt_fix_cos(rt_fix fix)
{
    rt_fix pi   = {1686629713, 2, 0}; /* PI     */
    rt_fix pi_2 = {1686629713, 3, 0}; /* PI * 2 */

    /*
     * transformation into intervall [0, 2 * PI]
     */
    fix = rt_fix_abs(fix);
    if(rt_fix_gtr(fix, pi_2)) {
        fix = rt_fix_div(fix, pi);
        if(rt_fix_tol(fix) & 0x00000001) {
            fix = rt_fix_add(rt_fix_mul(rt_fix_frac(fix), pi), pi);
        } else {
```

```
            fix = rt_fix_mul(rt_fix_frac(fix), pi);
        }
    }
    return rt_fix_sincos(fix, 1);

} /* End of rt_fix_cos */

/*
 * Calculate the arc cosine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              arc cosine from
 * Pre:
 *  fix e [-1, 1].
 * Out:
 *  A fixed point number which represents the approximated
 *  arc cosine of fix.
 */
rt_fix
rt_fix_acos(rt_fix fix)
{
    rt_fix pi_2 = {1686629713, 1, 0}; /* = 1.570796327          */
    rt_fix x_2 = fix;                 /* pow(fix, 2) and result */

    /*
     * all the else if alternatives are a bit of a
     * hack, because I couldn't find the bug.
     */
    if(rt_fix_gtr(rt_fix_abs(fix), rt_fix_1)) {
        fix = rt_fix_max;
        rt_fix_erraddset(&fix, RT_FIX_UNDEFINED);
        return fix;
    } else if(rt_fix_eq(fix, rt_fix_0)) {
        return pi_2;
    } else if(rt_fix_eq(fix, rt_fix_1)) {
        return rt_fix_0;
    } else if(rt_fix_eq(fix, rt_fix_neg(rt_fix_1))) {
        return rt_fix_pi;
    }
    x_2 = rt_fix_mul(x_2, x_2);
    x_2 = rt_fix_atan(rt_fix_div(rt_fix_sqrt(rt_fix_sub(rt_fix_1, x_2)), fix));
    if(rt_fix_gtr(fix, rt_fix_0)) {
        return x_2;
    }
    return rt_fix_add(rt_fix_pi, x_2);

} /* End of rt_fix_acos */

/*
 * Calculate the tangent of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the tangent from
 * Pre:
```

```
 *  |fix| <= 1024, to get a result with a precision of
 *  approximately 1E-4.
 * Out:
 *  A fixed point number which represents the
 *  approximated tangent of the fix
 */
rt_fix
rt_fix_tan(rt_fix fix)
{
    rt_fix cos, sin;

    cos = rt_fix_cos(fix);
    if(rt_fix_eq(cos, rt_fix_0)) {
        fix = rt_fix_max;
        rt_fix_erraddset(&fix, RT_FIX_UNDEFINED);
        return fix;
    }
    sin = rt_fix_sin(fix);
    return rt_fix_div(sin, cos);

} /* End of rt_fix_tan */

/*
 * Calculate the arc tangent of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              arc tangent from
 * Out:
 *  A fixed point number which represents the approximated
 *  arc tangent of fix.
 */
rt_fix
rt_fix_atan(rt_fix fix)
{

    rt_fix x_2, x_4, result;
    rt_fix p[] =
    {
        {1699194571, 4, 0},           /* = 12.6599861 */
        {1709716073, 3, 0}            /* =  6.3691887 */
    };
    rt_fix q[] =
    {
        {1699194571, 4, 0},           /* = 12.6599861 */
        {1421246360, 4, 0}            /* = 10.5891113 */
    };
    rt_fix pi_2 = {1686629713, 1, 0}; /* = PI / 2      */
    rt_fix pi_4 = {1686629713, 0, 0}; /* = PI / 4      */
    rt_fix i_l  = {889516852,  0, 0}; /* = sqrt(2) - 1 */
    rt_fix i_r  = {1296121037, 2, 0}; /* = sqrt(2) + 1 */
    short sign  = rt_fix_sign(fix);

    fix = rt_fix_abs(fix);
    /*
```

```
     * Hack, because I could not find the error.
     */
    if(rt_fix_eq(fix, rt_fix_1)) {
        if(sign) {
            return pi_4;
        } else {
            return rt_fix_neg(pi_4);
        }
    }
    if(rt_fix_leq(fix, i_l)) {
        x_2 = rt_fix_mul(fix, fix);
        x_4 = rt_fix_mul(x_2, x_2);
        result = rt_fix_mul(fix,
                   rt_fix_div(
                     rt_fix_add(p[0], rt_fix_mul(p[1], x_2)),
                     rt_fix_add(rt_fix_add(q[0], rt_fix_mul(q[1], x_2)), x_4)));
    } else if(rt_fix_lss(fix, i_r)) {
        fix = rt_fix_div(
                rt_fix_sub(fix, rt_fix_1),
                rt_fix_add(fix, rt_fix_1)
              );
        x_2 = rt_fix_mul(fix, fix);
        x_4 = rt_fix_mul(x_2, x_2);
        result = rt_fix_mul(fix,
                   rt_fix_div(
                     rt_fix_add(p[0], rt_fix_mul(p[1], x_2)),
                     rt_fix_add(rt_fix_add(q[0], rt_fix_mul(q[1], x_2)), x_4)));
        result = rt_fix_add(pi_4, result);
    } else {
        fix = rt_fix_div(rt_fix_1, fix);
        x_2 = rt_fix_mul(fix, fix);
        x_4 = rt_fix_mul(x_2, x_2);
        result = rt_fix_mul(fix,
                   rt_fix_div(
                     rt_fix_add(p[0], rt_fix_mul(p[1], x_2)),
                     rt_fix_add(rt_fix_add(q[0], rt_fix_mul(q[1], x_2)), x_4)));
        result = rt_fix_sub(pi_2, result);
    }
    if(sign) {
        return rt_fix_neg(result);
    } else {
        return result;
    }
} /* End of rt_fix_atan */

/*
 * Calculate fix raised to the power of e.
 * In:
 *  rt_fix fix  the fixed point number to be raised to the
 *              power of e.
 * Out:
 *  A fixed point number which represents the result of
 *  the calculation.
 */
```

```
rt_fix
rt_fix_exp(rt_fix x)
{
    int i;                              /* counter variable                  */
    int index = 21;                     /* index in exponent table           */
    rt_fix frac_result = rt_fix_0;      /* result of exp(rt_fix_frac(x))     */
    rt_fix frac_part   = rt_fix_0;      /* result of exp(rt_fix_frac(x))     */
    rt_fix int_part    = rt_fix_0;      /* result of exp(x - rt_fix_frac(x)) */
    rt_fix _x1;                         /* internal help variable            */
    rt_fix a[] =                        /* help coefficients                 */
    {
        {2147483219, 0, 0},  /* = 0.9999998 */
        {1073741824, 1, 0},  /* = 1.0000000 */
        {1073755353, 0, 0},  /* = 0.5000063 */
        {357915516,  0, 0},  /* = 0.1666674 */
        {89410482,   0, 0},  /* = 0.0416350 */
        {17888109,   0, 0},  /* = 0.0083298 */
        {3090873,    0, 0},  /* = 0.0014393 */
        {438087,     0, 0},  /* = 0.0002040 */
    };
    rt_fix e_n[] = {
        {0,          1, 0},  /* = exp(-21) */
        {2,          1, 0},  /* = exp(-20) */
        {6,          1, 0},  /* = exp(-19) */
        {16,         1, 0},  /* = exp(-18) */
        {44,         1, 0},  /* = exp(-17) */
        {120,        1, 0},  /* = exp(-16) */
        {328,        1, 0},  /* = exp(-15) */
        {892,        1, 0},  /* = exp(-14) */
        {2427,       1, 0},  /* = exp(-13) */
        {6597,       1, 0},  /* = exp(-12) */
        {17933,      1, 0},  /* = exp(-11) */
        {48747,      1, 0},  /* = exp(-10) */
        {132510,     1, 0},  /* = exp(-9)  */
        {360200,     1, 0},  /* = exp(-8)  */
        {979125,     1, 0},  /* = exp(-7)  */
        {2661539,    1, 0},  /* = exp(-6)  */
        {7234815,    1, 0},  /* = exp(-5)  */
        {19666267,   1, 0},  /* = exp(-4)  */
        {53458457,   1, 0},  /* = exp(-3)  */
        {145315153,  1, 0},  /* = exp(-2)  */
        {395007542,  1, 0},  /* = exp(-1)  */
        {1073741824, 1, 0},  /* = exp(0)   */
        {729683222,  3, 0},  /* = exp(1)   */
        {991742321,  4, 0},  /* = exp(2)   */
        {673958782,  6, 0},  /* = exp(3)   */
        {916004956,  7, 0},  /* = exp(4)   */
        {622489906,  9, 0},  /* = exp(5)   */
        {846051501,  10, 0}, /* = exp(6)   */
        {574951605,  12, 0}, /* = exp(7)   */
        {781440250,  13, 0}, /* = exp(8)   */
        {1062087416, 14, 0}, /* = exp(9)   */
        {721763231,  16, 0}, /* = exp(10)  */
        {980977937,  17, 0}, /* = exp(11)  */
```

```
        {666643625,  19, 0}, /* = exp(12)  */
        {906062626,  20, 0}, /* = exp(13)  */
        {615733393,  22, 0}, /* = exp(14)  */
        {836868447,  23, 0}, /* = exp(15)  */
        {568711073,  25, 0}, /* = exp(16)  */
        {772958488,  26, 0}, /* = exp(17)  */
        {1050559506, 27, 0}, /* = exp(18)  */
        {713929203,  29, 0}, /* = exp(19)  */
        {970330390,  30, 0}, /* = exp(20)  */
        {2147483647, 31, 0}  /* = exp(21)  */
    };

    if(rt_fix_toi(x) > 20) {
        x = rt_fix_max;
        rt_fix_erraddset(&x, RT_FIX_OV);
        return x;
    }
    if(rt_fix_toi(x) < -20) {
        x = rt_fix_0;
        rt_fix_erraddset(&x, RT_FIX_UV);
        return x;
    }
    frac_part = rt_fix_modf(x, &int_part);
    if(int_part._value != 0) {
        index = rt_fix_toi(int_part) + 21;
    }
    /*
     * compute exp(frac(x))
     */
    if(frac_part._value != 0) {
        _x1 = rt_fix_1;
        for(i = 0; i < 8; i++) {
            frac_result = rt_fix_add(frac_result, rt_fix_mul(a[i], _x1));
            _x1 = rt_fix_mul(_x1, frac_part);
        }
    } else {
        return e_n[index];
    }
    return(rt_fix_mul(frac_result, e_n[index]));

} /* End of rt_fix_exp */

/*
 * Calculate the logarithm to the base of two.
 * In:
 *  rt_fix x    the fixed point number to calculate
 *              the logarithm to the base of two from.
 * Out:
 *  - A fixed point number which represents the result of
 *    the calculation.
 *  - If x is negative the logarithm to the base of two of
 *    the absolute value of x is calculated.
 *  - If x is zero rt_fix_min is returned.
 */
```

```
rt_fix
rt_fix_log2(rt_fix x)
{
    int i;
    unsigned short shift;
    rt_fix u[3];
    rt_fix result     = rt_fix_0;
    rt_fix rt_fix_0_5 = {1073741824, 0, 0}; /* = 0.5          */
    rt_fix help       = {1518500250, 0, 0}; /* = sqrt(2) / 2  */
    rt_fix a[] =
    {
        { 1549082653, 2, 0},                /* = 2.8853912903 */
        { 2064742460, 0, 0},                /* = 0.9614706323 */
        { 1286296854, 0, 0}                 /* = 0.5989786496 */
    };

    if(rt_fix_sign(x)) {
        x = rt_fix_abs(x);
        rt_fix_erraddset(&result, RT_FIX_UNDEFINED);
    }
    if(rt_fix_eq(x, rt_fix_0)) {
        /*
         *  x == 0
         */
        x = rt_fix_min;
        rt_fix_erraddset(&x, RT_FIX_UNDEFINED);
        return x;
    }
    if(rt_fix_eq(x, rt_fix_1)) {
        /*
         *  x = 1
         */
        return rt_fix_0;
    }
    if(rt_fix_gtr(x, rt_fix_1)) {
        /*
         *  x > 1
         */
        shift = x._iwl;
        x = rt_fix_shr(x, shift);
        u[0] = rt_fix_div(rt_fix_sub(x, help), rt_fix_add(x, help));
        u[1] = rt_fix_mul(rt_fix_mul(u[0], u[0]), u[0]);
        u[2] = rt_fix_mul(rt_fix_mul(u[1], u[0]), u[0]);
        for(i = 0; i < 3; i++) {
            result = rt_fix_add(result, rt_fix_mul(a[i], u[i]));
        }
        result = rt_fix_sub(result, rt_fix_0_5);
        rt_fix_init_i(&help, shift);
        result = rt_fix_add(result, help);
    } else if(rt_fix_geq(x, rt_fix_0_5)) {
        /*
         * 0.5 <= x < 1
         */
        u[0] = rt_fix_div(rt_fix_sub(x, help), rt_fix_add(x, help));
```

```
        u[1] = rt_fix_mul(rt_fix_mul(u[0], u[0]), u[0]);
        u[2] = rt_fix_mul(rt_fix_mul(u[1], u[0]), u[0]);
        for(i = 0; i < 3; i++) {
            result = rt_fix_add(result, rt_fix_mul(a[i], u[i]));
        }
        result = rt_fix_sub(result, rt_fix_0_5);
    } else {
        /*
         * 0 < x < 0.5
         */
        x = rt_fix_div(rt_fix_1, x);
        shift = x._iwl;
        x = rt_fix_shr(x, shift);
        u[0] = rt_fix_div(rt_fix_sub(x, help), rt_fix_add(x, help));
        u[1] = rt_fix_mul(rt_fix_mul(u[0], u[0]), u[0]);
        u[2] = rt_fix_mul(rt_fix_mul(u[1], u[0]), u[0]);
        for(i = 0; i < 3; i++) {
            result = rt_fix_add(result, rt_fix_mul(a[i], u[i]));
        }
        result = rt_fix_sub(result, rt_fix_0_5);
        rt_fix_init_i(&help, shift);
        result = rt_fix_neg(rt_fix_add(result, help));
    }
    return result;

} /* End of rt_fix_log2 */

/*
 * Calculate the natural logarithm of fix.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the natural logarithm from.
 * Out:
 *  - A fixed point number which represents the result of
 *    the calculation.
 *  - If fix is negative the natural logarithm of the
 *    absolute value of fix is calculated.
 *  - If fix is zero rt_fix_min is returned.
 */
rt_fix
rt_fix_log(rt_fix fix)
{
    rt_fix ln_2 = {1488522236, 0, 0}; /* = 0.69314718 */

    return rt_fix_mul(ln_2, rt_fix_log2(fix));

} /* End of rt_fix_log */

/*
 * Calculate the decimal logarithm of fix.
 * In:
 *  rt_fix fix  the fixed point number to calculate
 *              the decimal logarithm from.
 * Out:
```

```
 *  - A fixed point number which represents the result of
 *    the calculation.
 *  - If fix is negative the logarithm to the base of ten of
 *    the absolute value of fix is calculated.
 *  - If fix is zero rt_fix_min is returned.
 */
rt_fix
rt_fix_log10(rt_fix fix)
{
    rt_fix log_2 = {646456993, 0, 0}; /* = 0.301029995 */

    return rt_fix_mul(log_2, rt_fix_log2(fix));

} /* End of rt_fix_log10 */

/*
 * Calculate x raised to the power of y.
 * In:
 *  rt_fix fix  the fixed point number to be raised
 *              to the power of y.
 *  rt_fix y    the exponent.
 * Pre:
 *  If y is not an integral value then x must be positive.
 * Out:
 *  - A fixed point number which represents the result of
 *    the calculation.
 *  - If fix is negative and y is not an integral value then
 *    the absolute value of fix is used to calculate the
 *    result and RT_FIX_UNDEFINED is set in the error set
 *    of the result.
 */
rt_fix
rt_fix_pow(rt_fix fix, rt_fix y)
{
    if(rt_fix_lss(fix, rt_fix_0) && rt_fix_neq(rt_fix_frac(y), rt_fix_0)) {
        fix = rt_fix_abs(fix);
        fix = rt_fix_exp(rt_fix_mul(y, rt_fix_log(fix)));
        rt_fix_erraddset(&fix, RT_FIX_UNDEFINED);
        return fix;
    }
    return rt_fix_exp(rt_fix_mul(y, rt_fix_log(fix)));

} /* End of rt_fix_pow */

/*
 * Calculate the hyperbolic sine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              hyperbolic sine from.
 * Out:
 *  A fixed point number which represents the result of
 *  the calculation.
 */
rt_fix
```

```
rt_fix_sinh(rt_fix fix)
{
    rt_fix one_half = {1073741824, 0, 0}; /* = 0.5 */

    return rt_fix_mul(rt_fix_sub(rt_fix_exp(fix),
                                 rt_fix_exp(rt_fix_neg(fix))),
                      one_half);

} /* End of rt_fix_sinh */

/*
 * Calculate the hyperbolic cosine of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              hyperbolic cosine from.
 * Out:
 *  A fixed point number which represents the result of
 *  the calculation.
 */
rt_fix
rt_fix_cosh(rt_fix fix)
{
    rt_fix one_half = {1073741824, 0, 0}; /* = 0.5 */

    return rt_fix_mul(rt_fix_add(rt_fix_exp(fix),
                                 rt_fix_exp(rt_fix_neg(fix))),
                      one_half);

} /* End of rt_fix_cosh */

/*
 * Calculate the hyperbolic tangent of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to calculate the
 *              hyperbolic tangent from.
 * Out:
 *  A fixed point number which represents the result of
 *  the calculation.
 */
rt_fix
rt_fix_tanh(rt_fix fix)
{
    return rt_fix_div(rt_fix_sub(rt_fix_exp(fix),
                                 rt_fix_exp(rt_fix_neg(fix))),
                      rt_fix_add(rt_fix_exp(fix),
                                 rt_fix_exp(rt_fix_neg(fix))));

} /* End of rt_fix_tanh */

#ifdef RT_FIXMISC
/*
 * assign a fixed point number to another fixed point number without
 * changing the integer word lenght of the fixed point number
 * to which is assigned.
```

```
 * InOut:
 *  rt_fix *left    pointer to fixed point number which is assigned to
 * In:
 *  rt_fix right    fixed point number which is assigned to left
 * Pre:
 *  The largest value the fixed point number left can be assigned to
 *  is equal or greater than the fixed point number right.
 * Post:
 *  The fixed point number left has the same value as the fixed
 *  point number right.  The integer word length of the fixed
 *  point number left is not changed.
 * Out:
 *  0   on success
 *  1   on error
 */
short
rt_fix_assfix(rt_fix *left, rt_fix right)
{
    long abs = rt_fix_tol(rt_fix_floor(rt_fix_abs(right)));
    long rvalue = rt_fix_get_rep(right);

    if(!left) {
        return 1;
    }
    /*
     * handle assignment overflow, not enough iwl.
     */
    rt_fix_errmergeset(left, right);
    if(abs >= (1 << left->_iwl)) {
        if(rt_fix_tol(right) >= 0) {
            left->_value = RT_FIXMAX;
            rt_fix_erraddset(left, RT_FIX_OV);
        } else {
            left->_value = RT_FIXMIN;
            rt_fix_erraddset(left, RT_FIX_OV);
        }
        return 1;
    }
    if(left->_iwl > right._iwl) {
        rvalue >>= (left._iwl - right->_iwl);
    }
    left->_value = rvalue;

    return 0;

} /* End of rt_fix_assfix */

/*
 * assign a integer number to a fixed point number without
 * changing the integer word lenght of the fixed point number
 * to which is assigned.
 * InOut:
 *  rt_fix *left    pointer to fixed point number which is
 *                  assigned to
```

```
 * In:
 *  int i           integer number which is assigned to left
 * Pre:
 *  The largest value the fixed point number left can be assigned to
 *  is equal or greater than i.
 * Post:
 *  The fixed point number left has the value i. The integer word
 *  length of the fixed point number left is not changed.
 * Out:
 *  0   on success
 *  1   on error (overflow)
 */
short
rt_fix_assi(rt_fix *left, int i)
{
    if(!left) {
        return 1;
    }
    /*
     * handle assignment overflow, not enough iwl.
     */
    if(((i >= 0) && (i >= (1 << left->_iwl))) ||
       ((i <  0) && (-i >= (1 << left->_iwl))))
    {
        if(i >= 0) {
            left->_value = RT_FIXMAX;
            rt_fix_erraddset(left, RT_FIX_OV);
        } else {
            left->_value = RT_FIXMIN;
            rt_fix_erraddset(left, RT_FIX_OV);
        }
        return 1;
    }
    left->_value = i << (RT_FIXWL - left->_iwl);
    return 0;

} /* End of rt_fix_assi */

/*
 * assign a long number to a fixed point number without changing
 * the integer word length of the fixed point number to which
 * is assigned.
 * InOut:
 *  rt_fix *left    pointer to fixed point number which is
 *                  assigned to
 * In:
 *  long l          long number which is assigned to left
 * Pre:
 *  The largest value the fixed point number left can be assigned to
 *  is equal or greater than l.
 * Post:
 *  The fixed point number left has the value l. The integer word
 *  length of the fixed point number left is not changed.
 * Out:
```

```
 *  0   on success
 *  1   on error (overflow)
 */
short
rt_fix_assl(rt_fix *left, long l)
{
    if(!left) {
        return 1;
    }
    /*
     * handle assignment overflow, not enough iwl.
     */
    if(((l >= 0) && (l >= (1 << left->_iwl))) ||
       ((l < 0)  && (-l >= (1 << left->_iwl))))
    {
        if(l >= 0) {
            left->_value = RT_FIXMAX;
            rt_fix_erraddset(left, RT_FIX_OV);
        } else {
            left->_value = RT_FIXMIN;
            rt_fix_erraddset(left, RT_FIX_OV);
        }
        return 1;
    }
    left->_value = l << (RT_FIXWL - left->_iwl);
    return 0;

} /* End of rt_fix_assl */

#ifndef __KERNEL__
/*
 * assign a double number to a fixed point number.
 * InOut:
 *  rt_fix *left    pointer to fixed point number which is
 *                  assigned to
 * In:
 *  double d        double number which is assigned to left
 * Pre:
 *  The largest value the fixed point number left can be assigned to
 *  is equal or greater than the d.
 * Post:
 *  The fixed point number left has the value d. The integer word
 *  length of the fixed point number left is not changed.
 * Out:
 *  0   on success
 *  1   on error (overflow)
 */
short
rt_fix_assd(rt_fix *left, double d)
{
    if(!left) {
        return 1;
    }
    /*
```

```
     * handle assignment overflow, not enough iwl.
     */
    if(((d >= 0.) && (((long)d) >= (1 << left->_iwl))) ||
       ((d < 0.)  && (-((long)d) >= (1 << left->_iwl))))
    {
        if(d >= 0.) {
            left->_value = RT_FIXMAX;
            rt_fix_erraddset(left, RT_FIX_OV);
        } else {
            left->_value = RT_FIXMIN;
            rt_fix_erraddset(left, RT_FIX_OV);
        }
        return 1;
    }
    left->_value = (long) (d * (1 << (RT_FIXWL - left->_iwl)));
    return 0;

} /* End of rt_fix_assd */
#endif /* __KERNEL__ */

/*
 * query integer word length of a fixed point number.
 * In:
 *  rt_fix fix  the fixed point number to query the
 *              integer word length from
 * Out:
 *  The integer word length of fix.
 */
unsigned short
rt_fix_get_iwl(rt_fix fix)
{
    return fix._iwl;

} /* End of rt_fix_get_iwl */

/*
 * query internal representation
 * In:
 *  rt_fix fix  the fixed point number to query the
 *              internal representation from
 * Out:
 *  The internal representation of fix.
 */
long
rt_fix_get_rep(rt_fix fix)
{
    return fix._value;

} /* End of rt_fix_get_rep */

/*
 * convert a fixed point number into its string representation.
 * The string representation has the following format:
 *  internal_value  integer-word-length
```

```
 *     = long           = unsigned short
 * In:
 *  rt_fix fix  the fixed point number to print
 *  char *str_rep   the string to print the representation
 *                  into.
 * Pre:
 *  str_rep is not NULL and its length is equal or greater than
 *  RT_FIX_STR_REP_SIZE.
 * Post:
 *  str_rep contains the string representation of the fixed point
 *  number fix.
 * Out:
 *  0   on success
 *  1   on error
 */
short
rt_fix_toa(rt_fix fix, char *str_rep)
{
    if(str_rep) {
        sprintf(str_rep, "%ld %hu", fix._value, fix._iwl);
        return 0;
    }
    return 1;

} /* End of rt_fix_toa */

/*
 * convert a given string representation of a fixed point number into
 * a fixed point number.
 * In:
 *  const char *str_rep  the string representation of a fixed point
 *                       number.
 *  rt_fix *result      the fixed point number to store the result of
 *                      the conversion into.
 * Pre:
 *  str_rep and result are not NULL and str_rep contains a valid string
 *  representation of a fixed point number.
 * Post:
 *  The fixed point number *result contains the fixed point number
 *  described by string str_rep.
 * Out:
 *  0   on success
 *  1   on error
 */
short
rt_fix_atofix(rt_fix *result, const char *string_rep)
{
    char *end1_ptr, *end2_ptr;
    long tmp_value;
    long tmp_iwl;

    if(!result || !string_rep) {
        return 1;
    }
```

```
#ifdef __KERNEL__
    tmp_value = (long)simple_strtoul(string_rep, &end1_ptr, 10);
    if(string_rep == end1_ptr) {
        return 1;
    }
    tmp_iwl = (int)simple_strtoul(end1_ptr, &end2_ptr, 10);
    if(end1_ptr == end2_ptr) {
        return 1;
    }
#else
    tmp_value = strtol(string_rep, &end1_ptr, 10);
    if(string_rep == end1_ptr) {
        return 1;
    }
    tmp_iwl = strtol(end1_ptr, &end2_ptr, 10);
    if(end1_ptr == end2_ptr) {
        return 1;
    }
#endif /* __KERNEL__ */
    if(tmp_iwl > RT_FIXWL) {
        return 1;
    }
    rt_fix_init_mask(result, tmp_value, (unsigned short)tmp_iwl);
    return 0;

} /* End of rt_fix_atofix */

/*
 * Set the integer word length of a fixed point number.
 * InOut:
 *  rt_fix *fix pointer to the fixed point number which
 *              is manipulated
 * In:
 *  unsigned short iwlen  the new integer word length
 * Post:
 *  The integer word length of the fixed point number is
 *  set to iwlen.
 * Out:
 *  0   on success
 *  1   on error
 */
short
rt_fix_set_iwl(rt_fix *fix, unsigned short iwlen)
{
    if(!fix) {
        return 1;
    }
    fix->_iwl = iwlen;
    return 0;

} /* End of rt_fix_set_iwl */

#endif /* RT_FIXMISC */
```

```
#ifdef __KERNEL__
/*
 * Standard Routines needed for loadable modules.
 */
int
init_module()
{

    return 0;

} /* End of init_module */

void
cleanup_module(void)
{

    return;

} /* End of cleanup_module */
#endif /* __KERNEL__ */
```