**Floating-point arithmetic can be expensive if you're using an integer-only processor. But floating-point values can be manipulated as integers, asa less expensive alternative.**

One advantage of using a high-level language is the native support of floating-point math. This simplifies the task of programming and increases the accuracy of calculations. When used in a system that includes a hardware floating-point math unit, this can be a very powerful feature. However, if floating-point math is used on a microprocessor that supports only integer math, significant overhead can be incurred to emulate floating point, both in ROM size and in execution time. The alternative, used by assembly language programmers for years, is to use fixed-point math that is executed using integer functions. This article will discuss a method to implement fixed-point math in C.

Fixed-point math typically takes the form of a larger integer number, for instance 16 bits, where the most significant eight bits are the integer part and the least significant eight bits are the fractional part. Through the simple use of integer operations, the math can be efficiently performed with very little loss of accuracy. Unfortunately, C does not provide native support for fixed-point math. The method presented in this article is essentially the same as in assembly language and is described in the following steps.

**Initialization**  
The first step in the fixed-point math algorithm is to initialize the system. This step defines the parameters used later in the development of the system. To illustrate each step, assume that your application requires a variable that ranges from 0 to 1 with a granularity of 0.01, another variable family that ranges from 0 to 100 with a granularity of 0.1, and a third variable family that ranges from -1,000 to 1,000 with a granularity of 0.01. The steps are performed as follows.

First, determine the maximum absolute value M that you wish to calculate for each class of fixed-point variable. The value of M for each example requirement is 1, 100, and 1,000, respectively.

Second, calculate the number of bits x required to store this number such that 2*x* M 2*x*-1.

If the number is to be signed, add 1 to *x*. For our example requirements, *x* is 1, 7, and 11, respectively.

Then, determine the granularity G that is required. The example requirements define the granularity and no further calculation is needed.

Next, calculate the number of bits y required to provide this granularity such that 1/2*y* G 1/2*y*-1. For our example requirements, this is 7 (1/128 = .0078125), 4 (1/16 = .0625), and 7.

Finally, the minimum number of bits required is *x*+*y*. Round this sum up to either 8, 16, or 32. This sum will be referred to later as the FULLSIZEINT. If this sum is greater than the maximum integer size in your system, then floating-point numbers are required and will probably be more efficient. The results for our example requirements for all the initial parameters are shown in Table 1. The resultant columns show the actual ranges and granularity available after the bit requirements are complete.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Table 1 Example initialization results** | | | | | | | | |
| **No.** | **Range** | **M** | **x** | **Resultant  range** | **Granularity** | **y** | **Resultant  granularity** | **Number of  bits required** |
| 1 | 0–1 | 1 | 1 | 0–1 | 0.01 | 7 | 0.0078125 | 8 |
| 2 | 0–100 | 100 | 7 | 0–127 | 0.1 | 4 | 0.0625 | 16 |
| 3 | -1000– | 1000 | 11 | -1024–1023 | 0.01 | 7 | 0.0078125 | 32 |
|  | 1000 |  |  |  |  |  |  |  |

**Definition**  
The second step in the algorithm is to create the definitions for C that are required to implement the variables. This consists of typedefs and macros, with the possibility that a function may need to be developed.

**Typedef**  
Definition of the fixed-point algorithm requires a typedef for each fixed-point variable type that will be used by the system. This typedef is a union with an embedded structure as follows. The structure assumes that the compiler assigns the bits in an integer from most significant to least significant. If this is not the case, reverse the declaration of the structure members integer and fraction in the structure. This typedef is not portable, and is best put in a unique header file:

**typedef union FIXEDX\_Ytag {  
    FULLSIZEINT full;  
    struct partX\_Ytag {  
        FULLSIZEINT integer: x;  
        FULLSIZEINT fraction:   
            FULLSIZEINT-x;  
    } part;  
} FIXEDX\_Y;**

FULLSIZEINT is either **long**, **int**, **short**, or **char** and either signed or unsigned and cannot be longer than the maximum length integer available, or the structure will not work. If FULLSIZEINT is longer than this maximum, then floating point needs to be used.

Notice that the fractional portion of the structure is calculated as FULLSIZEINT-*x*, instead of using *y* as the size of the member. Since a complete integer is available, I have chosen to increase the granularity of the system in order to decrease the error of the calculations. In your application, you may wish to increase the integer member in order to provide a method to check for math overflow.

**Macros**  
After the typedefs have been declared, macros need to be defined. Substitute the actual value for the equations using *X* and *Y* in each of the following macros.

Define a macro to convert a value into the fixed-point variable. A is the integer part and B is the decimal part, expressed as normal decimal. These values must be hard-coded constants so the preprocessor and compiler can completely resolve the macro into an integer. If either A or B are variables, then the macro will actually generate floating-point code and eliminate the savings of this algorithm. Check the results of the compile in the listing to make sure that the compiler functioned properly. (To do this, you must have the compiler interlace the assembly code with the C statements.)

**#define FIXEDX\_YCONST(A,B) (FULL-SIZEINT)((A<<Y) + ((B + 1/(2^(Y+1)))\*2^Y))**

Define macros to perform multiplication and division of the fixed-point variables:

**#define MULTX\_Y(A,B) (FULL-     SIZEINT+1)(A.full\*B.full+     2^(Y-1))>>Y #define DIVX\_Y(A,B) (FULL-     SIZEINT+1)((A.full<<Y+1)/     B.full)+1)/2**

where FULLSIZEINT+1 is the next largest integer over *X*+*Y*. If FULLSIZEINT is the largest available integer, then either floating point must be used, or a subroutine is required for multiply and divide.

Listing 1 shows the definitions for one of our example requirements sets. In this file, I first define the integer typedefs so the code that is written is portable. Next, the typedefs and macros are defined.

**Listing 1: Example typedef and macros**

**\*    Range 0-1.9921875  
\*    Granularity 0.0078125**

**typedef union FIXED1\_7tag {  
    unsigned char full;  
    struct part1\_7tag {  
    unsigned char fraction: 7;  
    unsigned char integer: 1;  
    } part;  
} FIXED1\_7;**

**#define FIXED1\_7CONST(A,B) (unsigned char)((A<<7) + ((B + 0.00390625)\*128))  
#define MULT1\_7(A,B) (unsigned short)(A.full\*B.full+64)>>7  
#define DIV1\_7(A,B)(unsigned short)((A.full<<8)/B.full)+1)/2**

Listing 2 shows the multiply and divide routines used for the 32-bit example, assuming that a 64-bit integer is not available in the system, since that cannot be done in a macro. Notice that in the divide routine, I am using a floating-point number for the calculation. If your system requires division, the amount of memory used cannot be reduced this way, but the add, subtract, compare, and multiply routines will still execute faster than the comparable floating-point routines.

**Listing 2: Multiply and divide for 32-bit variables**

**int MULT11\_21(FIXED11\_21 a, FIXED11\_21 b)  
{  
    int temp,result;  
    char sign = 0;  
      
    if (a.full < 0)  
    {  
        sign = 1;  
        a.full = -a.full;  
    }  
    if (b.full < 0)  
    {  
        sign^ = 1;  
        b.full = -b.full;  
    }**

**result = (((a.full & 0x0000FFFF) \* (b.full & 0x0000FFFF))+1048576)>>21;  
    result = result + ((((a.full>>16) \* (b.full & 0x0000FFFF))+16)>>5);  
    result = result + ((((b.full>>16) \* (a.full & 0x0000FFFF))+16)>>5);  
    temp = (a.full>>16) \* (b.full>>16);  
    result = result + (temp<<11);  
    return (result \* -sign);  
}**

**int DIV11\_21(FIXED11\_21 a, FIXED11\_21 b)  
{  
    double temp;  
    FIXED11\_21 result;  
    unsigned char sign = 0;**

**temp = (double)a.full/(double)b.full;  
    if (temp<0)  
    {  
        temp = -temp;  
        sign = 1;  
    }  
    result.part.integer = temp;  
    result.part.fraction = ((temp-result.part.integer)\*4194304 + 1)/2;  
    result.part.integer \*= -sign;  
    return (result.full);  
}**

**Usage**  
When using the fixed-point variables in an application, first declare variables using the new typedef. Next, use the new variables in the application as defined in the following paragraphs.

When adding, subtracting, or performing logical comparisons on two variables with the same-resolution fixed-point numbers, use the .full portion of the union and perform straight integer arithmetic. Listing 3 shows a small routine that performs each operation and prints the result for the number of type **FIXED1\_7**. In this routine, the numbers are added and the result is printed to the display. A set of comparisons are then done to determine if a subtraction would work, and if so, the subtraction is performed and the results written to the display. If a subtraction would result in an unsigned underflow, an error is displayed.

**Listing 3: Simple math on same granularity variables**

**void Test1\_7(FIXED1\_7 a, FIXED1\_7 b)  
{  
&  
nbsp;   FIXED1\_7 temp;**

**printf("Results of operations on 1\_7 variables\n");  
    temp.full = a.full + b.full;  
    printf("Addition result is %d.%2.2d\n", temp.part.integer,  
        (temp.part.fraction\*100+64)/128);  
    if (a.full < b.full)  
    {  
        printf("a is less than b. Subtraction overflows.\n");  
    }  
    if (a.full == b.full)  
    {  
        printf("a is the same as b. Result = 0.\n");  
    }  
    if (a.full > b.full)  
    {  
        temp.full = a.full - b.full;  
        printf("Subtraction result is %d.%2.2d\n", temp.part.integer,   
        (temp.part.fraction\*100+64)/128);  
    }  
}**

When adding, subtracting, or performing logical comparisons on two variables with different-resolution fixed-point numbers, first scale the different resolution number to the same resolution as the result. Listing 4 shows a small routine that performs some mixed operations and prints the results. This routine performs the same function as Listing 3.

**Listing 4: Simple math on different granularity variables**

**void Test7\_9(FIXED7\_9 a, FIXED1\_7 b)  
{  
    FIXED7\_9 temp;**

**printf("\nResults of operations on 7\_9 and 1\_7 variables\n");  
    temp.full = a.full + (b.full<<2);  
    printf("Addition result is %d.%1.1d\n", temp.part.integer,  
        (temp.part.fraction\*10+256)/512);  
    if (a.full < (b.full<<2))  
    {  
        printf("a is less than b. Subtraction overflows.\n");  
    }  
    if (a.full == (b.full<<2))  
    {  
        printf("a is the same as b. Result = 0.\n");  
    }  
    if (a.full > (b.full<<2))  
    {  
        temp.full = a.full - (b.full<<2);  
        printf("Subtraction result is %d.%1.1d\n", temp.part.integer,  
         (temp.part.fraction\*10+256)/512);  
    }  
}**

When multiplying or dividing variables of the same-resolution fixed-point numbers, use the macros previously defined. Listing 5 shows a small routine that performs each operation and prints the result. This routine simply performs the multiply and divide without checking, then displays the results. With multiply and divide, you cannot pass the full value, so you must pass the structure name.

**Listing 5: Multiplication and division on same granularity variables**

**void Test7\_9\_X(FIXED7\_9 a, FIXED7\_9 b)  
{  
    FIXED7\_9 temp;**

**printf("\nResults of multiply and divide on 7\_9 variables.\n");  
    temp.full = MULT7\_9(a,b);  
    printf("Multiply result is %d.%1.1d\n", temp.part.integer,  
        (temp.part.fraction\*10+256)/512);  
    temp.full = DIV7\_9(a,b);  
    printf("Divide result is %d.%1.1d\n", temp.part.integer,  
        (temp.part.fraction\*10+256)/512);  
}**

When multiplying or dividing variables of different-resolution fixed-point numbers, use the same macro as the result and scale the operands as with addition and subtraction. Listing 6 shows a small routine that performs an operation like this and prints the results.

**Listing 6: Multiplication and division on different granularity**

**void Test11\_21(FIXED11\_21 a, FIXED7\_9 b)  
{  
    FIXED11\_21 temp;**

**printf("\nResults of multiply and divide on 11\_21 and 7\_9 variables.\n");  
    temp.full = b.full << 12;  
    temp.full = MULT11\_21(a,temp);  
    printf("Multiply result is %d.%2.2d\n", temp.part.integer,  
        (temp.part.fraction\*100+1048576)/2097152);  
    temp.full = b.full << 12;  
    temp.full = DIV11\_21(a,temp);  
    printf("Divide result is %d.%2.2d\n", temp.part.integer,  
        (temp.part.fraction\*100+1048576)/2097152);  
}**

The value can be displayed using the integer and fractional portions of the structure.

**Words of caution**  
When these algorithms are implemented, a few areas of caution need to be addressed. If they aren't, erroneous results may be obtained.

The C language does not check integer arithmetic for overflows or underflows. The programmer must take care to either prevent or check for these conditions. This is usually accomplished by using a subroutine, possibly in assembly language, for each math operation and checking the limits. However, it can also be accomplished by adding bits to the integer portion, and limiting to a smaller number after each math operation.

The compiler may perform signed integer arithmetic in a subroutine and may not provide a tremendous benefit.

A rounding error may be injected in the result due to the value of the least significant bit. Since the expectation is an LSB of 0.1, and in fact the value is something less, an error of 1 LSB is injected that can compound during mathematical operations. For example, using our FIXED1\_7 example, 0.02 is represented as a full number of 3, and 0.06 is represented as a full number of 8. The sum should be 0.08, or a full number of 10. The actual result is 11, which is closer to 0.09. To limit this error, use a larger number such as 16 bits, and increase the granularity from 7 bits to 15 bits.

Fixed-point math provides a small, fast alternative to floating-point numbers in situations where small rounding errors are acceptable. After implementing the algorithms described in this article, your application will be able to harness the power of C and still retain the efficiency of assembly.

**Q** is a [fixed point](http://en.wikipedia.org/wiki/Fixed-point_arithmetic) number format where the number of [fractional](http://en.wikipedia.org/wiki/Fraction_(mathematics)) [bits](http://en.wikipedia.org/wiki/Bit) (and optionally the number of [integer](http://en.wikipedia.org/wiki/Integer) bits) is specified. For example, a Q15 number has 15 fractional bits; a Q1.14 number has 1 integer bit and 14 fractional bits. Q format is often used in hardware that does not have a floating-point unit and in applications that require [constant resolution](http://en.wikipedia.org/wiki/Fixed-point_arithmetic).

|  |
| --- |
| * [5 External links](http://en.wikipedia.org/wiki/Q_(number_format)#External_links) |

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Q_(number_format)&action=edit&section=1)**] Characteristics**

Because Q format numbers are fixed point, they can be stored and operated on as integers. The number of fractional bits and the underlying integer size are chosen on an application-specific basis, depending on the range and resolution needed.

The notation used is Q*m*.*n*, where:

* Q designates that the number is in Q format notation — the [Texas Instruments](http://en.wikipedia.org/wiki/Texas_Instruments) representation for signed fixed-point numbers.
* *m* (optional; default=0) is the number of bits used to designate the two's complement integer portion of the number, exclusive of the sign bit.
* *n* is the number of bits used to designate the two's complement fractional portion of the number, i.e. the number of bits to the right of the binary point.

Note that the most significant bit is always designated as the sign bit (the number is stored as a [two's complement](http://en.wikipedia.org/wiki/Two%27s_complement) number). Representing a signed fixed-point data type in Q format therefore always requires *m*+*n*+1 bits to account for the sign.

For a given Q*m*.*n* format, using an *m*+*n*+1 bit signed integer container with *n* fractional bits:

* its range is [-2*m*, 2*m* - 2-*n*]
* its resolution is 2-*n*

For example, a Q14.1 format number:

* requires 14+1+1 = 16 bits
* its range is [-214, 214 - 2−1] = [-16384.0, +16383.5] = [0x8000, 0x8001 … 0xFFFF, 0x0000, 0x0001 … 0x7FFE, 0x7FFF]
* its resolution is 2−1 = 0.5

Unlike [floating point](http://en.wikipedia.org/wiki/Floating_point), the resolution will remain constant over the entire range.

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Q_(number_format)&action=edit&section=2)**] Conversion**

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Q_(number_format)&action=edit&section=3)**] Float to Q**

To convert a number from [floating point](http://en.wikipedia.org/wiki/IEEE_754) to Q*m*.*n* format:

1. Multiply the floating point number by 2*n*
2. Round to the nearest integer

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Q_(number_format)&action=edit&section=4)**] Q to Float**

To convert a number from Q*m*.*n* format to floating point:

1. Convert the number to floating point as if it were an integer
2. Multiply by 2−*n*

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Q_(number_format)&action=edit&section=5)**] Math operations**

Q numbers are a ratio of two integers: the numerator is kept in storage, the denominator is equal to 2*n*.

Consider the following example:

The Q8 denominator equals 28 = 256

1.5 equals 384/256

384 is stored, 256 is inferred because it is a Q8 number.

If the Q number's base is to be maintained (*n* remains constant) the Q number math operations must keep the denominator constant.

\textstyle\frac{num1}{d} + \frac{num2}{d} = \frac{num1+num2}{d}

\textstyle\frac{num1}{d} - \frac{num2}{d} = \frac{num1-num2}{d}

(\textstyle\frac{num1}{d} \times \frac{num2}{d}) \times d = \frac{num1\times num2}{d}

(\textstyle\frac{num1}{d} / \frac{num2}{d})/d = \frac{num1/num2}{d}

Because the denominator is a power of two the multiplication can be implemented as an [arithmetic shift](http://en.wikipedia.org/wiki/Arithmetic_shift) to the left and the division as an arithmetic shift to the right; on many processors shifts are faster than multiplication and division.

To maintain accuracy the intermediate multiplication and division results must be double precision and care must be taken in [rounding](http://en.wikipedia.org/wiki/Rounding) the intermediate result before converting back to the desired Q number.

Using [C](http://en.wikipedia.org/wiki/C_(programming_language)) the operations are (note that here, Q refers to the fractional part's number of bits) :

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Q_(number_format)&action=edit&section=6)**] Addition**

signed int a,b,result;

result = a+b;

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Q_(number_format)&action=edit&section=7)**] Subtraction**

signed int a,b,result;

result = a-b;

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Q_(number_format)&action=edit&section=8)**] Multiplication**

// precomputed value:

#define K (1 << (Q-1))

signed int a, b, result;

signed long int temp;

temp = (long int) a \* (long int) b; // result type is operand's type

// Rounding; mid values are rounded up

temp += K;

// Correct by dividing by base

result= temp >> Q;

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Q_(number_format)&action=edit&section=9)**] Division**

signed int a,b,result;

signed long int temp;

// pre-multiply by the base (Upscale to Q16 so that the result will be in Q8 format)

temp = (long int)a << Q;

// So the result will be rounded ; mid values are rounded up.

temp = temp+b/2;

result = temp/b;

**Binary scaling** is a [computer programming](http://en.wikipedia.org/wiki/Computer_programming) technique used mainly by embedded [C](http://en.wikipedia.org/wiki/C_programming_language), [DSP](http://en.wikipedia.org/wiki/Digital_signal_processing) and [assembler](http://en.wikipedia.org/wiki/Assembly_language) programmers to perform a pseudo [floating point](http://en.wikipedia.org/wiki/Floating_point) using [integer](http://en.wikipedia.org/wiki/Integer) arithmetic. It is both faster and more accurate than directly using floating point instructions, however care must be taken not to cause an [arithmetic overflow](http://en.wikipedia.org/wiki/Arithmetic_overflow).

A position for the virtual 'binary point' is taken, and then subsequent arithmetic operations determine the resultants 'binary point'.

Binary points obey the mathematical laws of [exponentiation](http://en.wikipedia.org/wiki/Exponentiation).

To give an example, a common way to use integer maths to simulate floating point is to multiply the coefficients by 65536.

This will place the binary point at B16.

For instance to represent 1.2 and 5.6 floating point real numbers as B16 one multiplies them by 216 giving

78643 and 367001

Multiplying these together gives

28862059643

To convert it back to B16, divide it by 216.

This gives 440400B16, which when converted back to a floating point number (by dividing again by 216, but holding the result as floating point) gives 6.71999. The correct floating point result is 6.72.

The scaling range here is for any number between 65535.9999 and -65536.0 with 16 bits to hold fractional quantities (of course assuming the use of a 64 bit result register). Note that some computer architectures may restrict arithmetic to 32 bit results. In this case extreme care must be taken not to overflow the 32 bit register. For other number ranges the binary scale can be adjusted for optimum accuracy.

**Re-scaling after multiplication**

The example above for a B16 multiplication is a simplified example. Re-scaling depends on both the B scale value and the word size. B16 is often used in 32 bit systems because it works simply by multiplying and dividing by 65536 (or shifting 16 bits).

Consider the Binary Point in a 32 bit word thus:

0 1 2 3 4 5 6 7 8 9

X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X

Placing the binary point at

* 0 gives a range of -1.0 to 0.999999.
* 1 gives a range of -2.0 to 1.999999
* 2 gives a range of -4.0 to 3.999999 and so on.

When using different B scalings the complete B scaling formula must be used.

Consider a 32 bit word size, and two variables, one with a B scaling of 2 and the other with a scaling of 4.

1.4 @ B2 is 1.4 \* (2wordsize-2-1) == 1.4 \* 2 ^ 29 == 0x2CCCCCCD

Note that here the 1.4 values is very well represented with 30 fraction bits! A 32 bit [real number](http://en.wikipedia.org/wiki/IEEE_floating-point_standard) has 23 bits to store the fraction in. This is why B scaling is always more accurate than floating point of the same word size. This is especially useful in [integrators](http://en.wikipedia.org/wiki/Integrator) or repeated summing of small quantities where [rounding error](http://en.wikipedia.org/wiki/Rounding_error) can be a subtle but very dangerous problem, when using floating point.

Now a larger number 15.2 at B4.

15.2 @ B4 is 15.2 \* (2 ^ (wordsize-4-1)) == 15.2 \* 2 ^ 27 == 0x7999999A

Again the number of bits to store the fraction is 28 bits. Multiplying these 32 bit numbers give the 64 bit result 0x1547AE14A51EB852

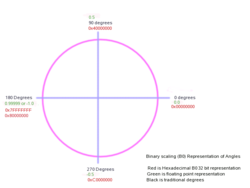
This result is in B7 in a 64 bit word. Shifting it down by 32 bits gives the result in B7 in 32 bits.

0x1547AE14

To convert back to floating point, divide this by (2^(wordsize-7-1)) == 21.2800000099

Various scalings maybe used. B0 for instance can be used to represent any number between -1 and 0.999999999.

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Binary_scaling&action=edit&section=2)**] Binary angles**

[](http://en.wikipedia.org/wiki/File:Binary_angles.png)

[http://bits.wikimedia.org/skins-1.5/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Binary_angles.png)

Binary Angles Diagram

Binary angles are mapped using B0, with 0 as 0 degrees, 0.5 as 90 (or π⁄2), −1.0 or 0.9999999 as 180 (or π) and −0.5 as 270 (or 3π⁄2). When these binary angles are added using normal [twos complement](http://en.wikipedia.org/wiki/Twos_complement) mathematics the rotation of the angles is correct, even when crossing the sign boundary (this of course does away with checks for angle ≥ 360 when handling normal degrees[[1]](http://en.wikipedia.org/wiki/Binary_scaling" \l "cite_note-0)).

The terms Binary Angular Measurement System (BAMS) [[2]](http://en.wikipedia.org/wiki/Binary_scaling#cite_note-1)[[3]](http://en.wikipedia.org/wiki/Binary_scaling#cite_note-ship-2) and 'brads' (binary radians) refer to implementations of binary angles. They find use in robotics, navigation[[4]](http://en.wikipedia.org/wiki/Binary_scaling#cite_note-3), computer games[[5]](http://en.wikipedia.org/wiki/Binary_scaling#cite_note-4), digital sensors[[6]](http://en.wikipedia.org/wiki/Binary_scaling#cite_note-5) and weapons system's digital communications[[3]](http://en.wikipedia.org/wiki/Binary_scaling#cite_note-ship-2) Binary angles may be thought of as the fractional part of an angle when expressed in units of [turn](http://en.wikipedia.org/wiki/Turn_(geometry)).