Micro Digital

GoFast® 8096 and Z80 Floating-Point Library User's Guide

October 21, 2009

by Harry Ohlson

Now maintained by
Micro Digital Associates Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

1 What Is GoFast?

1.1 A Floating-Point Library

GoFast is a floating-point library. It is a **soft** library, for processors that do not offer floating-point support in hardware. It is **complete**: no other floating-point routines are needed. It complies with the **IEEE 754** standard. However, the exception handling has been simplified a little (see section 4.1), mostly to make the product simple to use in embedded systems. GoFast goes to great pains to provide good **precision**, and to give mathematically **correct** answers even when the standards are silent. GoFast will run in **flash** memory: data and code are separate, there's no modification of code or constants, and there's no run-time initialization. It uses very little stack space.

This manual covers the versions of GoFast that were hand-written for pre-ANSI C compilers. (The main branch of the GoFast line is computer-generated and strictly ANSI, as documented in the GoFast Floating Point Library User's Guide.) These old compilers, mostly for 8-bit processors, provide only a limited set of math functions, and no double-precision at all. GoFast supplies the routines that were originally included, and adds the double-precision support.

To achieve reentrancy, the multitasker has to save and restore a few bytes (the floating-point accumulator) at each task switch.

Please see the readme files in the source code for detailed information related to the tools and other notes.

1.2 A Fast One

Most importantly, GoFast is **fast**. Replacing the native floating-point library with GoFast might cut timings by 20% for simple functions such as add or multiply, and by 75% in transcendentals such as the tangent. You could even see an occasional 90%, but there would be something wrong with the original routine then.

How can we make claims like these? Surely the people who supply the compiler do their best to produce a good floating-point library? Yes they do – their practical best. The floating-point routines are typically written in C and operate on floating-point variables. These algorithms are relatively simple, easily found on the Web or in books, and efficient in a floating-point unit. They get heavy when all floating-point is simulated.

GoFast performs all calculations using **integers**. The first thing done is the separation of the exponent and the mantissa; the last is their recombination. Because the mantissa has 64 bits, good precision comes as a bonus. The algorithms can get intricate – and you don't find them on the Web – but they have been thoroughly tested over the years.

1.3 Definitions

Floating point is a method of representing numeric values (integers and non-integers) in a computer. It uses three fields for this:

- The **sign** tells whether the number is positive or negative.
- The **exponent** tells where the decimal point goes.
- The **mantissa** (also called the significand) gives the digits.

To get the actual value of the number, you raise 2 to the power of the exponent and multiply this with the mantissa. (For details such as bias and scaling, see the IEEE 754 document.)

In the IEEE 754 standard, **single-precision** numbers take up 32 bits, **double-precision** numbers twice that. The useful **range** for singles is approximately 10^{-38} to 10^{38}, for doubles 10^{-308} to 10^{308}. The relative **precision** (typical rounding error in one arithmetic operation) is of the order of 10^{-7} for singles, 10^{-16} for doubles.

1.4 Notations

In the following text, these symbols denote different kinds of variables:

d1, **d2**, **d3**	double-precision number
f1, **f2**, **f3**	single-precision number
si	standard integer (16 or 32 bits usually)
li	long integer (32 bits)
ul	unsigned long integer (32 bits)
NaN	not-a-number, an invalid floating-point value
INF	infinity, an overflowed floating-point value

2 Included Functions

2.1 Intrinsic Functions

These are functions called by the C compiler to handle simple floating-point operations such as add or compare. The names of the intrinsics depend on the compiler in question, and there's even some (small) variation in what exact functions are needed. All these functions also have a private GoFast name, for testing and documentation purposes. The following tables give the GoFast name.

function type	operation	generated call	notes
double arithmetic	`d1 = d1 + d2`	`d1 = dpadd(d1,d2)`	
	`d1 = d1 – d2`	`d1 = dpsub(d1,d2)`	
	`d1 = d1 * d2`	`d1 = dpmul(d1,d2)`	
	`d1 = d1 / d2`	`d1 = dpdiv(d1,d2)`	
single arithmetic	`f1 = f1 + f2`	`f1 = fpadd(f1,f2)`	
	`f1 = f1 – f2`	`f1 = fpsub(f1,f2)`	
	`f1 = f1 * f2`	`f1 = fpmul(f1,f2)`	
	`f1 = f1 / f2`	`f1 = fpdiv(f1,f2)`	
conversion	`d1 = f1`	`d1 = double(f1);`	`float to double`
	`f1 = d1`	`f1 = single(d1);`	`double to float`
	`d1 = si`	`d1 = dfloat(si);`	`integer to double`
	`si = d1`	`si = dfix(d1);`	`double to integer`
	`f1 = si`	`f1 = float(si);`	`integer to float`
	`si = f1`	`si = fix(f1);`	`float to integer`
comparison	`d1 :: d2`	`si = dpcmp(d1, d2)`	`0x80 0xff 0 1`
	`f1 :: f2`	`si = fpcmp(f1, f2)`	`0x80 0xff 0 1`

The **dpcmp** and **fpcmp** returns are:

0x80 no meaningful comparison (few compilers care)
0xff argument 1 < argument 2
0 arguments equal
1 argument 1 > argument 2

2.2 User Functions

This group comprises the math functions. The names of the user-level functions are of course fixed (**sin**, **sqrt** and so on), but the compiler often adds something to this, perhaps an underscore or two.

function type	user call	function performed
double, simple	d1 = dpsqrt(d1)	square-root
single, simple	f1 = fpsqrt(f1)	square-root
double, transc.	d1 = dpatn(d1)	arctangent
	d1 = dpcos(d1)	cosine
	d1 = dpexp(d1)	**e** to the power d1
	d1 = dpln(d1)	natural logarithm
	d1 = dplog(d1)	base 10 logarithm
	d1 = dpsin(d1)	sine
	d1 = dptan(d1)	tangent
	d1 = dpxtoi(d1, si)	d1 to the power si
single, transc.	f1 = fpatn(f1)	arctangent
	f1 = fpcos(f1)	cosine
	f1 = fpexp(f1)	**e** to the power f1
	f1 = fpln(f1)	natural logarithm
	f1 = fplog(f1)	base 10 logarithm
	f1 = fpsin(f1)	sine
	f1 = fptan(f1)	tangent
	f1 = fpxtoi(f1, si)	f1 to the power si

3 Testing

Different versions of GoFast offer different ways of testing the library. Please see the text files included with the product for details.

3.1 GFTEST

GFTEST will repeatedly prompt for a value and then display the results of various calculations.

3.2 FPTEST, DPTEST

These programs (the first for single precision, the second for double) are stack-based calculators. The allowed operations are:

value	Push the value entered onto the stack
+	Add the top-of-stack item to the next-on-stack item
-	Subtract the top-of-stack item from the next-on-stack item
_	Subtract the next-on-stack item from the top-of-stack item
*	Multiply the top-of-stack item by the next-on-stack item
/	Divide the next-on-stack item by the top-of-stack item
\	Divide the top-of-stack item by the next-on-stack item
?	Compare the top-of-stack item with the next-on-stack item
A	Perform the AINT operation to the top-of-stack item
C	Display the top-of-stack item using the binary-to-ASCII routine
Cnnn	Convert the value to floating point number using the ASCII-to-binary routine
FA	Arc-tangent of the top-of-stack item
FC	Cosine of the top-of-stack item
FE	e raised to the top-of-stack item power
FL	Common logarithm (base 10) of the top-of-stack item
FN	Natural logarithm (base e) of the top-of-stack item
FR	Square root of the top-of-stack item
FS	Sine of the top-of-stack item
FT	Tangent of the top-of-stack item
F^nn	Raise the top-of-stack item to the integer power given
Fnn	The integer given is floated to become the top-of-stack item
Gn	Get the number from register n (n = 1..9)
Hhh	The hexadecimal number become the top-of-stack item
I	Display the INT function of the top-of-stack item
M	Change precision mode (single/double)
Q	Exit to the operating system
R	Roll the four stack items upward
Sn	Store the top-of-stack number in register n (n = 1..9)
X	Exchange the top-of-stack item with the next-on-stack item

4 Technical Considerations

4.1 Exception Handling

GoFast makes no distinction between quiet and signaling not-a-numbers (NaNs). In an invalid operation, the answer is always a quiet NaN, 0x001FFFFFFFFF in double precision and 0x007FFFFF in single precision.

The GoFast routines support the IEEE 754 masked exception handling for overflows and invalid operations. An overflow is returned as the special value infinity, and an invalid operation is returned as the special value NaN.

No unmasked exceptions are supported; there are no exception interrupts. GoFast stores an error code into the byte variable FPERR. The values are: 3 for not-a-number, 2 for overflow and 1 for underflow.

4.2 Precision

The basic operations (add, subtract, multiply, divide, square root) and the conversions all use the IEEE 754 "round to nearest or even" rounding exactly. No other rounding modes are supported. These operations are IEEE exact.

The transcendental functions (which are not defined in IEEE 754) are correct to within two mantissa units. However, the trigonometric functions SIN, COS and TAN will lose precision in the argument reduction if the argument exceeds $\pi/2$.

4.3 Special Values

An overflow returns +INF or -INF, an underflow returns +0 or -0. If an argument is not-a-number (NaN), the result is NaN. The table below gives the GoFast result for some other special situations. It does not include cases that should not cause any confusion.

-	INF-INF = NaN
*	0*INF = NaN
/	0/0 = NaN
	INF/INF = NaN
sqrt	sqrt(-0) = -0
	sqrt(x<0) = NaN
ln/log	-INF if x=0
	NaN if x<0
sin/cos/tan	NaN if |x| >= 65536

Most likely, these pathological cases will be of no interest to anyone. It is not at all unusual to find a C library that returns questionable values for one or more.

4.4 Accuracy in Calculations

Floating-point calculations are in practice always inexact. This is easy to forget because just about everything else in programming is exact, and because the precision seldom becomes a problem. But you forget at your own peril.

There is nothing mysterious about the loss of precision; it's simply the nature of the thing. The following illustrates different faces of the inaccuracy.

4.4.1 Rounding

A floating-point number contains a fixed number of digits. Unless there are a lot of trailing zeroes, an arithmetic operation will very likely produce too many digits to fit in the same space. This of course happens even in normal decimal calculations, for instance:

```
  1234.567
+  12.34567
  1246.91267  ➔ 1246.913
```

Rounding errors as such are unlikely to become noticeable, but they can be enhanced by other effects. Some algorithms are notoriously prone to lose precision.

4.4.2 Base Conversion

Changing the base of a fractional number generally requires approximations. Any application that uses decimal input, decimal constants or decimal output has to perform base conversions. Consider the example

```
float f1;
f1 = 1.1;
printf("%.12f\n", f1);
```

This program will display the value 1.100000023842, not the exact 1.1. What happened?

The root of the problem is that 1 1/10 in base 2 is 1.0001100(1100), i.e. can't be represented exactly. The compiler creates a constant 1.1 with 24 bits:

1.000 1100 1100 1100 1100 1101

This value is obviously larger than 1.1 because we rounded up at bit 24. Printing the value with too many decimals (anything more that 7 in this case) will show the difference.

4.4.3 Difference between Large Numbers

Let's try the program

```
float f1, f2, f3;
f1 = 1234.0;
f2 = 1233.1;
f3 = f1 - f2;
printf("%lf\n", f3);
```

The result is 0.900024: off by quite a bit. The basic effect is the same as explained above: the required base conversion. But the relative error got enlarged in the subtraction of two almost equal numbers:

```
    1234.0 = 1001 1010 0100 0000 0000 0000
-   1233.1 = 1001 1010 0010 0011 0011 0011
     0.9 =                 1100 1100 1101
```

4.4.4 Irrational Numbers

Values such as sqrt(2) or sin(0.5) have no exact representation in any base. These can still be calculated "exactly" to the value that is mathematically correct considering the rounding rules. IEEE specifically requires an exact square-root, but says nothing about other functions. The GoFast square-root is of course exact.

You probably won't find an "exact" implementation of the transcendentals anywhere. The additional error should be of the same order as the rounding error.

4.4.5 Special Functions

As a rule, the relative error of a function is different than the relative error of the argument. In some cases this becomes important. Take the following code:

```
double d1, d2;
d1 = 1.1;
d3 = exp(100*d1);
```

The result will differ from exp(110) by quite a bit. This does not mean that exp(x) is inaccurate; it means that the original inaccuracy of x got magnified.

A point where a function approaches zero for a non-zero argument is especially tricky. As an example, log(0.999998) is close to twice log(0.999999). If your argument is only a little inexact, say due to rounding, the answer may be so wrong as to be meaningless. Again we need to remember that log(x) as such is not the culprit, it is not inaccurate.

The same warning applies whenever significant argument reduction is needed, such as the trigonometric functions for arguments much larger than π. Worst of all are cases where these two situations coincide: sin(1000π) for instance.

4.4.6 Conversion to Integer

ANSI C specifies that a floating-point number is converted to an integer using truncation: the decimals are discarded. This innocuous rule can cause surprises. Consider the program

```
int i1, i2;
i1 = 256;
i2 = (float)i1 / 2.56;
printf("%d\n", i2);
```

Certainly the correct answer is 100, but you can't count on this; the program as written is unstable. In some cases, the answer will keep jumping between 99 and 100, depending on the compilation options and the exact code used.

The root reason for the instability is not hard to see. The value 2.56 has to be rounded when it is converted to base 2. If this rounding is up, the division will give a value that is slightly less than 100. According to ANSI C rules, this becomes 99. If again 2.56 in base 2 is rounded down, the division will give slightly over 100, and truncates to 100.

IEEE 754 is a very rigorous standard; whether 2.56 is rounded up or down, surely it should be rounded the same way every time. How is it possible that two standard implementations give completely different results? Well, it really isn't. This is an interesting example of what happens when a standard meets an optimizing compiler. How the rounding is done depends on the number of bits in the constant. ANSI C says that a floating-point constant is **double**, and IEEE 754 rules this to have 53 binary digits. Unfortunately

1. Some compilers use **float** constants in float expressions. This difference may be enough to change the direction of the rounding.
2. Some compilers optimize out all divisions by a constant, using instead a multiplication with the inverse value. What happens to the rounding is anybody's guess.

4.4.7 Financial Calculations

You want to make absolutely sure your broker isn't cheating you, so you write a little program to check the commission. The first trade looks fine. The second trade looks fine. The third trade – caught him! Overcharged by a penny!

Well, not really. Financial rounding follows law and custom, knowing (and caring) nothing about IEEE 754 rounding. In some special cases, you have to round up. Even the usual "bank rounding" isn't quite the same as the IEEE default – though you'll have to look hard to catch the difference.

None of this means that there's a problem. Financial institutions just don't use floating-point math.

5 Processor Version Details

5.1 Intel 8096/80196

The GoFast library works with the Intel IC96 compiler. The native library has no double-precision routines, and the compiler will not generate any calls to such routines. When you install GoFast, you can start using double-precision, but you'll have to write the function calls explicitly. You'll find examples of this in the GoFast test files.

The native library lacks **asin**, **acos, atan2** and all the hyperbolics. Instead of **pow**, there's a function that raises a number to an integer power. GoFast will not add these missing functions. The implementation is compatible with the IEEE 754 standard, but it isn't really ANSI C, nor could it be.

GoFast for 80196 implements a floating-point accumulator (FAC) in read-write memory, so it isn't naturally reentrant. However, you get reentrancy by saving and restoring FAC (and a few other temporaries) in a context switch. A note included with the product gives the details.

5.2 Z80/Z180/64180

5.2.1 Compiler

This GoFast library is for the Archimedes/IAR C compiler. The native library has no double-precision routines, and the compiler will not generate any calls to such routines. When you install GoFast, you can start using double-precision, but you'll have to write the function calls explicitly. You'll find examples of this in the GoFast files.

The native library lacks **asin**, **acos, atan2** and all the hyperbolics. Instead of **pow**, there's a function that raises a number to an integer power. GoFast will not add these missing functions. The implementation is compatible with the IEEE 754 standard, but it isn't really ANSI C, nor could it be.

GoFast for the Z80 family implements a floating-point accumulator (FAC) in read-write memory, so it isn't naturally reentrant. However, you get reentrancy by saving and restoring FAC (and a few other temporaries) in a context switch. A note included with the product gives the details.

5.2.2 Test Environment

The product is shipped with instructions on how to test using the PatchLan HD64180 test system.

5.2.3 Timings

The following table shows the timings (in microseconds) for a few functions on the 6 MHz Z80. The given range is from a typical value to a maximum value.

Function	Single	Double
add	163 - 227	550 - 950
multiply	693 – 817	3350 - 3567
divide	905 - 1267	5583 - 8000
sin/cos	6667	42667
log	7167	37000
sqrt	2500	20333

6 References

ANSI/IEEE Standard 754-1985: Binary Floating-Point Arithmetic

W. Cody, W. Waite: Software Manual for the Elementary Functions, Prentice-Hall, 1980