Micro Digital

GoFast® 8051 Floating-Point Library User's Guide

August 17, 2010

© Copyright 2009-2010 Micro Digital Inc
© Copyright 1983-2009 Lantronix Inc

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

All rights reserved.

1 Introduction

Notes: FPAC refers to the single-precision floating point routines and DPAC refers to the double-precision routines. This manual and application notes were re-entered from a printed copy. Some introductory sections were recently written, and other edits have been made.

1.1 Purpose

GoFast is a software floating-point library for processors that do not offer floating-point support in hardware. It complies with the IEEE 754 standard. However, the exception handling has been simplified a little, mostly to make the product simple to use in embedded systems.

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. The floating-point routines provided with the compiler 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.

The GoFast floating point library includes FPAC and DPAC parts. FPAC provides floating point functions based on the IEEE single precision floating point format. DPAC extends the FPAC functionality to the IEEE double precision floating point format.

The library consists of the basic floating point operations (ADD/SUBTRACT, MULTIPLY, DIVIDE), data conversion routines (ASCII to/from floating point, integer to/from floating point), functions (sine, cosine, tangent, arctangent, common and natural logarithm, exponentiation of e, square root, and floating point to integer power). In addition, DPAC provides precision translation routines for conversion between single and double precision formats.

The library was designed to emphasize accuracy, source code clarity, code size efficiency, and execution speed. Wherever IEEE floating point standards exist, and when they are feasible to implement in software with respect to the scope and purpose of this package, FPAC and DPAC adhere to them.

1.2 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.

1.3 The IEEE Floating Point Format

The number format on which FPAC operates is the IEEE 754 single precision standard. Its representation, in bit form is:

S EEE EEEE	E MMM MMMM	MMMM MMMM	MMMM MMMM
byte 3	byte 2	byte 1	byte 0

“S” is the sign bit (1 if negative, 0 if positive). The “E” field is the two’s exponent. It is a two’s complement value biased by 127 (decimal. The “M” field is the 23-bit normalized mantissa. The most significant bit is always assumed to be 1, and so is not explicitly stored. This yields an effective precision of 24 bits.

The value of the floating point number described above is obtained by multiplying 2 raised to the power of the unbiased exponent, by the binary mantissa. The assumed bit of the binary mantissa (the most significant bit) has a value of 1.0, with the remaining bits providing a fractional value (i.e., the value of the mantissa is greater than or equal to 1.0 and less than 2.0).

Note that the four bytes of the floating point number are stored in lexicographic order. As noted in the architectural description of the 8051, the processor’s convention is that the least significant byte has the lowest address value. This means the sign/exponent byte is stored at a higher memory address than the mantissa bytes.

The dynamic range of the IEEE 754 single precision floating point format is +/- 1.175494E-38 to 3.402823E+38.

The number format on which DPAC operates is the IEEE 754 double precision standard. Its representation, in bit form, is:

S EEE EEEE	EEEE MMMM	MMMM MMMM	MMMM MMMM
byte 7	byte 6	byte 5	byte 4
MMMM MMMM	MMMM MMMM	MMMM MMMM	MMMM MMMM
byte 3	byte 2	byte 1	byte 0

“S” is the sign bit (1 if negative, 0 if positive). The “E” field is the two’s exponent. It is a two’s complement value biased by 1023 (decimal). The “M” field is the 52-bit normalized mantissa. The most significant bit is always assumed to be 1, and so is not explicitly stored. This yields an effective precision of 53 bits.

The value of the floating point number described above is obtained by multiplying 2 raised to the power of the unbiased exponent, by the binary mantissa. The assumed bit of the binary mantissa (the most significant bit) has a value of 1.0, with the remaining bits providing a fractional value (i.e., the value of the mantissa is greater than or equal to 1.0 and less than 2.0).

Note that the eight bytes of the floating point number are stored in lexicographic order. As noted in the architectural description of the 8051, the processor’s convention is that the least significant byte has the lowest address value. This means the sign/exponent byte is stored at a higher memory address than the mantissa bytes.

The dynamic range of the IEEE 754 double precision floating point format is +/- 2.2250 73858 50720D-308 to 1.7976 93134 86231D+308.

1.4 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 **π**/2.

1.5 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.

1.6 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, 0x0008000000000000 in double precision and 0x00400000 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.

1.7 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.

1.7.1 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:

$$\begin{array}{rl} & 1234.567 \\ + & \underline{\quad 12.34567} \\ & 1246.91267 \;\rightarrow\; 1246.913 \end{array}$$

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.

1.7.2 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.

1.7.3 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:

$$\begin{array}{crr} & 1234.0 = & 1001\ 1010\ 0100\ 0000\ 0000\ 0000 \\ - & \underline{1233.1 =} & \underline{1001\ 1010\ 0010\ 0011\ 0011\ 0011} \\ & 0.9 = & 1100\ 1100\ 1101 \end{array}$$

1.7.4 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.

1.7.5 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(1000**π**) for instance.

1.7.6 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.

1.7.7 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.

1.8 Resource Requirements

1.8.1 Memory Conventions

The 8051 FPAC/DPAC routines work with four and eight byte floating point values that reside in either the external data memory or in the program memory (read-only constants). External operands are addressed using the MOVX instruction with DPTR or the MOVC instruction. The routines maintain a floating point accumulator (shared by single and double precision routines) in the on-chip data memory along with exception flags and some working storage. The conversion routines and functions produce temporaries that are held in external data memory.

1.8.2 Resource Requirements

The 8051 FPAC/DPAC routines use three types of 8051 resources; bit addressable on-chip data memory, on-chip data memory, and external data memory.

Either one or two bytes of bit addressable on-chip data memory is required. A single byte (called FACBIT) is used by the basic operations and an additional type (called CNVBIT) is used by the ASCII/binary conversion routines and/or the functions.

The basic single precision operations require 14 bytes of on-chip data memory. The basic double precision operations require 12 more bytes of on-chip data memory. The ASCII/binary conversion routines use one byte of on-chip data memory. A fully implemented single precision library will have an on-chip data memory requirement of 15 bytes while a fully implemented double precision library will require 27 bytes of on-chip data memory.

The ASCII/binary conversion routines use one external data memory temporary (four or eight bytes depending on precision). Functions may use up to four temporaries (again either four or eight bytes each), but one temporary may be overlaid on the conversion temporary making a maximum external data memory requirement of 16 bytes for single precision and 32 bytes for double precision.

1.9 Parameter Passing

A floating point value is passed to an FPAC/DPAC routine by placing its address in DPTR. If the value resides in program memory (a constant) instead of the external data memory, then the FACRNM bit must be set (for Rom Number).

The FPAC/DPAC routines maintain a Floating Point Accumulator (referred to as the “FAC”) in the on-chip data memory. A value will remain in the FAC until the user changes it either explicitly or implicitly. In most cases, the contents of the FAC is used as an operand by FPAC/DPAC routines. Thus, the contents of the FAC is a parameter that is implicitly passed to and returned from FPAC/DPAC routines. Binary operations take place as FAC <oper> [DPTR] → FAC. Unary functions are performed on the value in the FAC with the result returned in the FAC.

In general, FPAC/DPAC routines destroy the contents of the accumulator (called A or ACC), the B register, and the current register bank (R0 to R7). The value of the DPTR is returned unchanged (except for the ASCII to binary conversion routines which advance it as noted).

The FPAC/DPAC routines use some stack space for temporaries. The conversion routines and functions used temporaries in the external data memory. The amount of stack space required by the various routines, beyond the two bytes for the return, is noted in the discussion of the individual routines.

Word integer values are passed to and from FPAC/DPAC routines in the B:A (with B holding the most significant byte). Four byte integer values are passed in R7:R6:B:A (R7 is the most significant byte, A is the least significant byte). All integer values are in two’s complement form.

1.10 Franklin/Keil C Compiler Version

1.10.1 Compiler Details

GoFast is a drop-in replacement library for the Franklin/Keil 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 8051 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.

1.10.2 Timings

The following table shows the GoFast timings for a few functions on a 12 MHz 8051, in microseconds. The given range is from a typical value to a maximum value.

Functions	Single	Double
add	260 - 370	750 – 1100
multiply	450 - 560	1380 – 1530
divide	1070 - 1390	5300 – 6900
sin/cos	5050	23400
log	6000	23000
sqrt	2850	18500

2 Basic Floating Point Operations

2.1 LDFAC & LDFACD – Load Floating Point Accumulator (FAC)

These routines load floating point values from memory (pointed to by DPTR) into the floating point accumulator (FAC). If FACRNM is set, the value is taken from program memory. If FACRNM is clear, the value is taken from external data memory. LDFAC is a single precision routine while LDFACD is its double precision counterpart. The FACRNM bit is always cleared by LDFAC and LDFACD.

The FAC consists of three parts: its sign, its exponent, and its mantissa. The sign is held in a byte called FACSGN. The sign bit is replicated throughout this byte, so FACSGN's value is either 0 or OFFH (-1).

The exponent is held in a byte pair referred to as FACEXP. The exponent's bias is not removed while in the FAC. Single precision exponent values are zero-extended one byte to fill the double type FACEXP. Double precision exponent values are zero-extended 5 bits to fill FACEXP.

The mantissa is held in a series of bytes headed by the byte named FACMAN. In the case of single precision values in the FAC, the series of bytes is three long. A double precision value in the FAC uses seven bytes. Unlike the four or eight byte representation of the floating point number, the mantissa in the FAC explicitly represents what is called the implicit (or "j") bit of the floating point number's mantissa. The mantissa, as with the exponent, is right justified in the FAC's mantissa register.

2.2 STFAC & STFACD – Store Floating Point Accumulator (FAC)

The value in the FAC is compressed as required and stored in memory at the location indicated by DPTR.

2.3 FPADD & DPADD – Addition and Subtraction

The floating point value pointed to by DPTR (and FACRNM) is added to the floating point value in the FAC. The FPADD routine is used to sum single precision floating point numbers while the DPADD routine works with double precision floating point values.

Subtraction of two floating point values is accomplished by flipping the sign bit of the subtrahend, then calling the appropriate addition routine. If the subtrahend contains zero or NaN, the sign should not be complemented.

Both FPADD and DPADD use four bytes of stack space.

The following routines implement double precision floating point subtraction operations. The result is always left in the FAC, though either the FAC or the operand pointed to by DPTR may be the subtrahend. Note that the sign is not complemented if the value is zero or NaN.

```
DPSUB:      CALL DNGFAC            ; FAC = – FAC
            CALL DPADD             ; FAC = OPN – FAC
            BRA   DNGFAC           ; FAC = FAC – OPN
;
DPRSUB:     CALL DNGFAC            ; FAC = – FAC
            JMP   DPADD            ; FAC = OPN – FAC
;
;       Negate FAC
;
DNGFAC:     MOV   A, FACEXP – 0    ; Check zero
            MOV   B, FACEXP – 1
            CJNE  A, B, DNGF01     ; J/ FAC < > 0
            JZ    DNGF03           ; J/ FAC = 0 (NO NEGATION)
;
DNGF01:     CJNE  A, #007H, DNGF02 ; J/ FAC < > NaN
            MOV   A, B
            CJNE  A, #0FFH, DNGF02 ; J/ FAC < > NaN
            MOV   A, FACMAN – 0
            ANL   A, #00001111B    ; Strip implicit bit
            JNZ   DNGF03           ; J/ FAC = (NaN or INF)
;
DNGF02:     CPL   FACSGN           ; Flip sign
DNGF03      RET
```

2.4 FPMUL & DPMUL – Multiplication

The floating point value in the FAC is multiplied by the floating point value pointed to by DPTR (and FACRNM). The FPMUL routine works with single precision operands, the DPMUL routine processes double precision values.

Both FPMUL and DPMUL use two bytes of stack space.

The following routine squares the single precision floating point value addressed by DPTR (and FACRNM), leaving the result in the FAC.

```
FPSQ:       MOV   C, FACRMN        ; Save ROM/RAM bit
            MOV   HOLDBT, C
            CALL  LDFAC            ; FAC = Value
            MOV   C, HOLDBT        ; Get ROM/RAM bit
            MOV   FACRMN, C
            JMP   FPMUL                ; FAC = FAC * Value (square)
```

2.5 FPDIV, FPRDIV & DPDIV, DPRDIV – Division

Division of floating point values is performed by calling one of these division routines. The xPDIV routines use the value pointed to by DPTR (and FACRMN) as the divisor and the value in the FAC as the dividend, placing the result in the FAC (that is, FAC/[DPTR] → FAC). The xPRDIV routines use the value pointed to by DPTR (and FACRMN) as the dividend and the value in the FAC as the divisor, placing the result in the FAC (that is, [DPTR]/FAC → FAC). The FPxxxx routines work with single precision floating point values while the DPxxxx routines handle double precision floating point numbers.

All of FPxxxx and DPxxxx routines use four bytes of stack space.

The following routine reciprocates the single precision floating point in the FAC.

```
FPREC:      MOV   DPTR. #FPONE         ; DPTR points to 1.0
            SETB  FACRMN               ; in program memory
            JMP   FPRDIV               ; FAC = 1.0/ FAC

FPONE:      DB    000H, 000H, 080H, 03FH ; Single Precision 1.0
```

2.6 FPCMP & DPCMP – Comparison Routines

The floating point value in the FAC is compared to the floating point value pointed to by the DPTR (and FACRMN). The FPCMP routine is used to compare single precision floating point numbers while the DPCMP routine is used to compare double precision floating point values.

The comparison routines may use the addition/substraction routine (of the appropriate precision) if necessary. See section 6.0 for the results of comparing the special representations of +INF, -INF, and NaN with each other or standard point values.

Both routines implement a fuzz specification. The values are 20 bits for FPCMP (the symbolic constant FFUZZ) and 48 bits for DPCMP (the symbolic constant DFUZZ). The fuzz value indicates the number of bits which must be equal. For example, in the single precision case, if the result of the FAC minus the operand is at least 2^-20 times smaller than the large of the FAC and the operand, the values are considered equal even though there may be a slight difference in actual values.

The result of the comparison is returned in the A register as follows:

Comparison	A Register	
No comparison	080H	-128
FAC < OPN	0FFH	-1
FAC = OPN	000H	0
FAC > OPN	001H	1

A quick comparison result testing algorithm is:

1) Clear carry, rotate the accumulator left
2) If A < > 0 then
    a. If carry set then, FAC < OPN
    b. else (carry clear), FAC > OPN
3) else (A = 0)
    a. If carry set then, FAC does not compare
    b. else (carry clear), FAC = OPN

FPCMP and DPCMP may destroy the contents of the FAC. Both FPCMP and DPCMP use up to eight bytes of stack space.

2.7 FLOAT & DFLOAT – Integer to FP Value Conversion

A register resident integer is converted to a floating point value in the FAC by the float routines. The Float routine converts the two’s complement 16 bit integer in B:A (B holds the most significant byte) into a single precision floating point value in the FAC. The DFLOAT routine converts the two’s complement 32 bit integer in R7:R6:B:A (R7 holds the most significant byte, A holds the least significant byte) into a double precision value in the FAC.

No stack space is used by FLOAT. DFLOAT uses 2 bytes of stack space.

The routine below “floats” an eight bit integer value in A into a double precision floating point value in the FAC.

```
BDFLT:      MOV   B, #0              ; assume positive
            JNB   ACC.7, BDFL01      ; J/ value positive
            DEC   B                  ; Sign extend thru B
BDF01:      MOV   R6, B              ; Sign extend thru R6
            MOV   R7, B              ; Sign extend thru R7
            JMP   DFLOAT             ; FAC = DFLOAT (R7: R6: B: A)
```

2.8 INT, FIX & DINT, DFIX – FP Value to Integer Conversion

Two methods are provided to convert the floating point value in the FAC into a register resident integer. The single precision routines, INT and FIX, process single precision floating point values and return the resulting 16 bit two’s complement integer in B:A. The double precision floating point numbers into a 32 bit two’s complement integers in R7: R6:B: A.

The difference between the two methods, FIX and INT, is illustrated in the table below:

F.P. Value	FIX(value)	INT(value)
3.5	3	3
- 3.5	- 3	- 4

The result of a FIX operation is the argument value stripped of its fractional part. The result of an INT operation is the largest integer such that it is less than or equal to the argument value.

FIX does not use any stack space. INT uses 2 bytes of stack space. DFIX uses 2 bytes of stack space. DINT uses 4 bytes of stack space.

2.9 AINT & DAINT – Floating Point INT Function

The floating point value in the FAC is “INT-ed” in place by calling the routine AINT for single precision values or DAINT for double precision floating point numbers. While approximately the same function could be accomplished by using an INT then FLOAT sequence, AINT operations are considerable faster and of higher precision. Since this operation is performed in the floating point domain, the AINT routine works with 24 bit precision while the DAINT routine work with 53 bit precision.

In addition, arguments provided to AINT routine that are too large or invalid (NaN, +INF, -INF), are returned unchanged. INT routines will return a maximum magnitude integer of the appropriate sign these instances.

AINT and DAINT do not use any stack space.

3 Precision Conversion Routines

DPAC provides two precision conversion routines, SINGLE and DOUBLE, for conversion between the two floating point formats supported by FPAC/DPAC. The value in the FAC is converted.

3.1 SINGLE – Double to Single Precision Conversion Routine

The SINGLE routine converts the double precision floating point number in the FAC into a single precision floating point number in the FAC. The conversion is a round-to-nearest process. Double precision floating point values that are too large to represent in the single precision format overflow to infinity (INF) with an appropriate sign, while those values that are too small to represent underflow to zero. NaNs and INFs are carried through directly. In any event, no error flags are set by this conversion routine.

No stack space is used by the SINGLE routine.

3.2 Double – Single to Double Precision Conversion Routine

The Double routine converts the single precision floating point number in the FAC into a double precision floating point number in FAC. The conversion is a precision extension process (by setting the additional mantissa bits to zero). All single precision floating point values can be properly represented in the double precision format. NaNs and INFs are carried through directly. In any event, no error flags are set by this conversion routine.

No stack space is used by the DOUBLE routine.

4 ASCII Literal to/from Floating Point

4.1 ASCBIN & DASCBN – ASCII Literal to Floating Point Value

These routines convert an ASCII literal point to by DPTR into a floating point value in the FAC. DPTR points to the first character of the literal (in external data memory) to be interpreted. This first character of the literal must be either a minus sign (indicating a negative number), a decimal point, or a digit; these routines will not skip preceding blanks.

The first character of the literal field plus subsequent characters must form a valid decimal number, optionally followed immediately by "E", ("D" is also allowed by the double precision routine), signifying scientific notation. If an "E" is found, it may be followed by a plus or a minus sign, then one or two digits (the sign and digits indicating the power of ten scaling), the double precision routine allows up to three digits in the exponent field.

These routines will process characters until an improper character is found. This "improper" character may be a blank, comma, zero byte, etc. When the maximum allowed digits have been found after the "E", the next character is automatically improper.

A NaN error code is returned if an improper literal is encountered. Note that reaching an improper character is NOT an error condition; it can indicate the correct termination of the ASCII literal. Errors include no digits in the literal, no digits preceeding the "E", no digits following the "E". A literal that, when interpreted, is too large to represent is returned as infinity (INF). Literals that are too small to represent underflow to zero.

A floating point value is always returned in the FAC. In the case of an error, the value is what amounts to a best guess (or the value when things went awry).

The single precision routine (ASCBIN) uses 10 bytes of stack space while the double precision routines (DASCBN) use 14 bytes of stack space.

4.1.1 Example of ASCII Literals and Results of the Conversion

Input String	Value	Error	Returned pointer position
3.567,	3.567	No error	At ","
- .5	-0.5	No error	At byte after "5"
5e4	50000	No error	At byte after "4"
- .E4	0.0	Error	At "E"
3.1.2	3.1	No error	At second "."
4.2E 13	42.0	Error	At blank after "E"
-6E98	-INF	*Error	At byte after "8"
	0.0	Error	Pointer unchanged

^ initial pointer position

*: this is a valid double precision floating point value

4.2 BINASC & DBNASC – Floating Point Value to ASCII Literal

These routines are provided to convert the floating point value in the FAC into a zero byte terminated string of ASCII characters at the location pointed to by DPTR.

Both BINASC and DBNASC return DPTR unchanged. However, these routines perform a number of internal floating point operations which may destroy the value in the FAC. The single precision routine (BINASC) uses 13 bytes of stack space while the double precision routine (DBNASC) uses 16 bytes of additional stack space.

The first character of the ASCII literal produced is either a minus sign (for a negative value) or a blank, depending on the sign of the number being converted. Based on the value being converted, one of the following formats is selected:

4.2.1 BINASC – Single Precision

Form	Value Range		
0.n	0.1	to	0.9999999
N.n	1.0	to	9.999999
NN.n	10.0	to	99.99999
NNN.n	100.0	to	999.9999
NNNN.n	1000.0	to	9999.999
NNNNN.n	10000.0	to	99999.99
NNNNNN.n	100000.0	to	999999.9
NNNNNNNN.	1000000.0	to	9999999.

N.nE#dd	other valid number
0.	0 or underflow
+INF	positive infinity
-INF	negative infinity
NaN	not a number

4.2.2 DBNASC – Double Precision

Form	Value Range		
0.n	0.1	to	0.999999999999999
N.n	1.0	to	9.99999999999999
NN.n	10.0	to	99.9999999999999
NNN.n	100.0	to	999.999999999999
…	…	to	…
NNNNNNNNNNNNN.n	1000000000000.0	to	9999999999999.99
NNNNNNNNNNNNNN.n	10000000000000.0	to	99999999999999.9
NNNNNNNNNNNNNNN.	100000000000000.0	to	999999999999999.

N.nD#ddd	other valid number
0.	0 or underflow
+INF	positive infinity
-INF	negative infinity
NaN	not a number

Where N is a digit, n is the fractional part with trailing zeroes suppressed, “.” is a decimal point, “#” is either “+” or “-“, and “d” is a digit in the ten’s exponent. See section 6.0 for a description of the error conditions underflow, INF, and NaN.

The symbolic constant FDDIG specifies the number of digits the single precision routine will display. The symbolic constant FNDIG controls the initial conversion step (before the ASCII literal is formatted); the must be greater that FDDIG. As illustrated above, the release value of FNDIG is seven.

The corresponding symbolic constants for the double precision conversion routine are DDDIG and DNDIG. The release value of DNDIG is 15.

The output area is used for two purposes. It is first used to form an internal, intermediate literal that is reformatted for the second use, as the output literal. The minimum size for the single precision output area is FNDIG+10 bytes (17 bytes for the release version). The minimum size of the output area for the double precision routine is DNDIG+12 bytes (27 bytes for the release version).

5 API

5.1 Assembly API

The Assembly API provided in the library expects the arguments in the FAC and places the results in the FAC. (The FPXTOI and DPXTOI routines also use a 16-bit, two's complement integers in B: A; this value is destroyed during processing).

The single precision functions may use up to 12 bytes of stack space while the double precision functions may use up to 14 bytes of additional stack space.

All function results are correct to within one or two mantissa bits except for tangent near its discontiguous points and the logarithms very near 1.0 (though this is a representation problem of floating point values, not an algorithm error).

See the C API is implementation in files gf_dp*.a51 and gf_fp*.a51 for example use of the assembly API.

Single	Double	Description
FPATN	DPATN	Arctangent, range – PI/2 to + PI/2
FPCOS	DPCOS	Cosine (note: limited domain, see 6.0)
FPEXP	DPEXP	e raised to the power
FPLN	DPLN	Natural logarithm
FPLOG	DPLOG	Common logarithm
FPSIN	DPSIN	Sine (note: limited domain, see 6.0)
FPSQRT	DPSQRT	Square root
FPTAN	DPTAN	Tangent (note: limited domain, see 6.0)
FPXTOI	DPXTOI	Raise value to integer power

The code below illustrates the computation of one real root of a quadratic polynomial. The coefficients of the polynomial are assumed to reside at the memory locations named REALA, REALB, and REALC.

```
;
;  -------- External Data Memory
;
REALA       DS     4
REALB       DS     4
REALC       DS     4
FPTEMP      DS     4
;
;  -------- Program Memory
;        Calculate (-b + SQRT (b^2 – 4ac)) / 2a
;
         MOV     DPTR, #REALB             ; Calc b^2
```

```
        CALL    FPSQ                    ;  ( see FPMUL)
        MOV     DPTR, #FPTEMP
        CALL    STFAC                   ;  FPTEMP = b^2
;
        MOV     DPTR, #REALA
        CALL    LDFAC                   ; FAC = a
        MOV     DPTR, #REALC
        CALL    FPMUL                   ; FAC = ac
        MOV     DPTR, #FPFOUR
        SETB    FACRNM
        CALL    FPMUL                   ; FAC = 4ac
;
        MOV     DPTR, #FPTEMP
        CALL    FPRSUB                  ; reverse subtract
        CALL    FPSQRT                  ; FAC = SQRT (b^2 – 4ac)
        MOV     DPTR, #REALB
        CALL    FPSUB                   ; FAC = -b+SQRT (b^2 – 4ac)
;
        MOV     DPTR, #REALA
        CALL    FPDIV                   ; FAC = (-b+SQRT(b^2 – 4ac) / a
        MOV     DPTR, #FPTWO
        SETB    FACRNM
        CALL    FPDIV                   ; FAC = (-b+SQRT (b^2 – 4ac) / 2a
;
;       Constants (in program memory)
;
FPTWO:      DB      000H, 000H, 000H, 040H    ;       Single precision 2.0
FPFOUR:     DB      000H, 000H, 080H, 040H    ;       Single precision 4.0
```

5.2 C API

GoFast also has a C API, which can work with Keil C v5.02 compiler or later. We tested on Keil C v8.16. To use a different compiler, the API needs to be changed. The C API is implemented in files gf_dp*.a51 and gf_fp*.a51.

Since most C compilers for 8051 do not support a double type, the DOUBLE type is defined as a structure:

```
struct ieeedp
{
   unsigned int wrd[4];
};
typedef struct   ieeedp  DOUBLE;
```

Note: The GoFast interface assumes that all variables are in XDATA. It uses pointers to these variables for processing.

Single Precision

char cabs (char);	same as Keil math lib, can replace Keil's
int abs(int);	same as Keil math lib, can replace Keil's
long labs(long);	same as Keil math lib, can replace Keil's
float fabs (float);	same as Keil math lib, can replace Keil's
float fpabs(float);	same as Keil math lib, can replace Keil's
float fpsin(float);	
float sin(float);	same as Keil math lib, can replace Keil's
float fpcos(float);	
float cos(float);	same as Keil math lib, can replace Keil's
float fptan(float);	
float tan(float);	same as Keil math lib, can replace Keil's
float fpatan(float);	
float atan(float);	same as Keil math lib, can replace Keil's
float fpexp(float);	
float exp(float);	same as Keil math lib, can replace Keil's
float fplog10(float);	
float log10(float);	same as Keil math lib, can replace Keil's
float fplog(float);	
float log(float);	same as Keil math lib, can replace Keil's
float fpsqrt(float);	
float sqrt(float);	same as Keil math lib, can replace Keil's
float fpceil(float);	
float ceil(float);	same as Keil math lib, can replace Keil's
float fpfloor(float val);	
float floor(float val);	same as Keil math lib, can replace Keil's
float fppow(float, float);	
float pow (float, float);	same as Keil math lib, can replace Keil's
float fpadd(float, float);	also supports x+y, can replace Keil's
float fpsub(float, float);	also supports x-y, can replace Keil's
float fprsub(float, float);	
float fpmul(float, float);	also supports x*y, can replace Keil's
float fpdiv(float, float);	also supports x/y, can replace Keil's
float fprdiv(float, float);	
char fpcmp3(float x, float y);	-1:x<y; 0:x==y; 1:x>y; -128: NOT compare also supports x?y, can replace Keil's
void fpftoa(float, char*);	convert float to string
void ftoa(float, char*);	
float fpatof (char*);	convert string to float
float atof (char*);	
char ftoc(float);	convert float to char (8 bits)
char fptoc(float);	
char fpftoc(float);	
int ftoi(float);	convert float to int (16 bits)
int fptosi(float);	

int fpftoi(float);	
long ftol(float);	convert float to long (32 bits)
long fptoli(float);	
long fpftol(float);	
unsigned char ftouc(float);	convert float to unsigned char (8 bits)
unsigned char fptouc(float);	
unsigned char fpftouc(float);	
unsigned int ftoui(float);	convert float to unsigned int (16 bits)
unsigned int fptoui(float);	
unsigned int fpftoui(float);	
unsigned long ftoul(float);	convert float to unsigned long (32 bits)
unsigned long fptoul(float);	
unsigned long fpftoul(float);	
float ctof(char);	convert char (8 bits) to float
float ctofp(char);	
float fpctof(char);	
float sitofp(int);	convert int (16 bits) to float
float itof(int);	
float fpitof(int);	
float litofp(long);	convert long (32 bits) to float
float ltof(long);	
float fpltof(long);	
float uctof(unsigned char);	convert unsigned char (8 bits) to float
float uctofp(unsigned char);	
float fpuctof(unsigned char);	
float uitofp(unsigned int);	convert unsigned int (16 bits) to float
float uitof(unsigned int);	
float fpuitof(unsigned int);	
float ultofp(unsigned long);	convert unsigned long (32 bits) to float
float ultof(unsigned long);	
float fpultof(unsigned long);	

Double Precision

void dpadd (DOUBLE xdata *ag1,DOUBLE xdata *ag2,DOUBLE xdata *ans);
    ans = ag1 + ag2
void dpsub (DOUBLE xdata *ag1,DOUBLE xdata *ag2,DOUBLE xdata *ans);
    ans = ag1 - ag2
void dpmul (DOUBLE xdata *ag1,DOUBLE xdata *ag2,DOUBLE xdata *ans);
    ans = ag1 * ag2
void dpdiv (DOUBLE xdata *ag1,DOUBLE xdata *ag2,DOUBLE xdata *ans);
    ans = ag1 / ag2
void dppow (DOUBLE xdata *ag1, DOUBLE xdata *ag2,DOUBLE xdata *ans);
    ans = ag1 to pow ag2
char dpcmp3 (DOUBLE xdata *ag1,DOUBLE xdata *ag2);
    -1:x<y; 0:x==y; 1:x>y; -128: NOT compare

void dpsqrt (DOUBLE xdata *arg1,DOUBLE xdata *ans);	ans = sqrt(arg1)
void dpexp (DOUBLE xdata *arg1,DOUBLE xdata *ans);	ans = exp(arg1)
void dplog (DOUBLE xdata *arg1,DOUBLE xdata *ans);	ans = ln(arg1)
void dplog10 (DOUBLE xdata *arg1,DOUBLE xdata *ans);	ans = log10(arg1)
void dpsin (DOUBLE xdata *arg1,DOUBLE xdata *ans);	ans = sin(arg1)
void dpcos (DOUBLE xdata *arg1,DOUBLE xdata *ans);	ans = cos(arg1)
void dptan (DOUBLE xdata *arg1,DOUBLE xdata *ans);	ans = tan(arg1)
void dpatan (DOUBLE xdata *arg1,DOUBLE xdata *ans);	ans = atan(arg1)
void dpfloor (DOUBLE xdata *val, DOUBLE xdata *ans);	ans = dpfloor(arg1)
void dpceil (DOUBLE xdata *val, DOUBLE xdata *ans);	ans = dpcei(arg1)
void dpatod (char xdata *s,DOUBLE xdata *d);	string to double
void dpdtoa (DOUBLE xdata *d,char xdata *s);	double to string
void dptofp (DOUBLE xdata *d1, float xdata *f1);	convert double to float
void fptodp (float xdata *f1, DOUBLE xdata *d1);	convert float to double
char dtoc (DOUBLE xdata *arg1);	convert double to char (8 bits)
char dptoc (DOUBLE xdata *arg1);	
char dpdtoc (DOUBLE xdata *arg1);	
int dtoi (DOUBLE xdata *arg1);	convert double to int (16 bits)
int dptosi (DOUBLE xdata *arg1);	
int dpdtoi (DOUBLE xdata *arg1);	
long dtol (DOUBLE xdata *arg1);	convert double to long (32 bits)
long dptoli (DOUBLE xdata *arg1);	
long dpdtol (DOUBLE xdata *arg1);	
unsigned char dtouc (DOUBLE xdata *arg1);	convert double to unsigned char (8 bits)
unsigned char dptouc (DOUBLE xdata *arg1);	
unsigned char dpdtouc (DOUBLE xdata *arg1);	
unsigned int dtoui (DOUBLE xdata *arg1);	convert double to unsigned int (16 bits)
unsigned int dptoui (DOUBLE xdata *arg1);	
unsigned int dpdtoui (DOUBLE xdata *arg1);	
unsigned long dtoul (DOUBLE xdata *arg1);	convert double to unsigned long (32 bits)
unsigned long dptoul (DOUBLE xdata *arg1);	
unsigned long dpdtoul (DOUBLE xdata *arg1);	
void ctod (char, DOUBLE xdata *ans);	convert float to char (8 bits)
void dpcod (char, DOUBLE xdata *ans);	
void dpctod (char, DOUBLE xdata *ans);	
void itod (int, DOUBLE xdata *ans);	convert float to int (16 bits)
void sitodp (int, DOUBLE xdata *ans);	
void dpitod (int, DOUBLE xdata *ans);	
void ltod (long, DOUBLE xdata *ans);	convert float to long (32 bits)
void litodp (long, DOUBLE xdata *ans);	
void dpltod (long, DOUBLE xdata *ans);	
void uctod (unsigned char, DOUBLE xdata *ans);	convert float to unsigned char (8 bits)
void uctodp (unsigned char, DOUBLE xdata *ans);	
void dpuctod (unsigned char, DOUBLE xdata *ans);	
void uitod (unsigned int, DOUBLE xdata *ans);	convert float to unsigned int (16 bits)

```
void uitodp (unsigned int, DOUBLE xdata *ans);
void dpuitod (unsigned int, DOUBLE xdata *ans);
void ultod (unsigned long, DOUBLE xdata *ans); convert float to unsigned long (32 bits)
void ultodp (unsigned long, DOUBLE xdata *ans);
void dpultod (unsigned long, DOUBLE xdata *ans);
```

Use kfptodp and kdptofp instead of dptofp and fptodp for Keil 5.02 or later.

```
#define kfptodp(fp,dp) _flipflop ((fp)); fptodp ((fp), (dp)); _flipflop ((fp))
#define kdptofp(dp,fp) dptofp ((dp), (fp)); _flipflop ((fp))
```

6 Error Conditions

The IEEE 754 Floating Point standard defines several special representations. Signed infinity is represented as a sign bit, an exponent field of all ones, and a mantissa field of all zeroes. The result of an invalid operation is called Not-a-Number (NaN) and is represented as an exponent field of all ones and a non-zero mantissa field (the sign bit is insignificant but is generally set).

After FPAC and DPAC operations, the low order bits in FACBIT are set to one of the following values:

FACBIT	Description	Result Value
0	No error	per operation
1	Underflow	zero
2	Overflow	+INF or –INF
3	Invalid Operation	NaN

6.1 Addition

FAC

OPN

	0	number	+INF	-INF	NaN
0	0	number	+INF	-INF	NaN
number	number	**	+INF	-INF	NaN
+INF	+INF	+INF	+INF	NaN	NaN
-INF	-INF	-INF	NaN	-INF	NaN
NaN	NaN	NaN	NaN	NaN	NaN

****:** result could be 0 (possibly an underflow), a number, +INF, or –INF

6.2 Multiplication

FAC

OPN

	0	number	+INF	-INF	NaN
0	0	0	NaN	NaN	NaN
number	0	**	+/-INF	-/+INF	NaN
+INF	NaN	+/-INF	+INF	-INF	NaN
-INF	NaN	-/+INF	-INF	+INF	NaN
NaN	NaN	NaN	NaN	NaN	NaN

****:** result could be 0 (possibly an underflow), a number, +INF, or –INF

6.3 Division

DIVISOR

DIVIDEND

	0	number	+INF	-INF	NaN
0	NaN	0	0	0	NaN
number	+/-INF	**	0*	0*	NaN
+INF	+INF	+/-INF	NaN	NaN	NaN
-INF	-INF	-/+INF	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN

***:** underflow
: result could be 0 (possibly an underflow), a number, +INF, or –INF

6.4 Comparison

FAC

OPN

	0	number	+INF	-INF	NaN
0	=	< or >	>	<	*
number	< or >	<, =, >	>	<	*
+INF	<	<	*	<	*
-INF	>	>	>	*	*
NaN	*	*	*	*	*

<: FAC is less than operand
=: FAC is equal to operand (within FUZZ specification)
>: FAC is greater than operand
*: FAC does not compare to operand

6.5 Functions

Function

ARG

	ATN	EXP	LN/LOG	SQRT	Trig
-INF	PI/2	0*	NaN	NaN	NaN
-num	number	**	NaN	NaN	***
0	0	1	-INF	0	0 or 1
+num	number	****	number	number	***
+INF	PI/2	+INF	+INF	+INF	NaN
NaN	NaN	NaN	NaN	NaN	NaN

*: underflow
**: 0 (possibly underflow) or a number less than 1.0
***: 0, num, +INF, -INF, or NaN (if magnitude >= 65536)
****: number > 1.0 or +INF

6.6 X to I function

Integer Power

		-/Odd	-/Even	0	+/Even	+/Odd
OPN	-INF	0*	0*	NaN	+INF	-INF
	-num	**	***	1.0	***	**
	0	+INF	+INF	NaN	0	0
	+num	***	***	1.0	***	***
	+INF	0*	0*	NaN	+INF	+INF
	NaN	NaN	NaN	NaN	NaN	NaN

*: underflow
**: 0 (an underflow), negative number, or –INF
***: 0 (an underflow), positive number, or +INF

7 Routine Sizes and Execution Times

Note: These measurements were made sometime in the past, but are probably reliable. They will be updated in a future revision.

7.1 8051 FPAC Routine Sizes and Execution Times

The sizes shown below are in bytes of program space. The execution time is given in CPU cycles for the 8051. Each CPU cycle is 12 oscillator periods.

7.1.1 Basic Operations Set – FPAC

Name	Size	Exec. typ	Times max	Comments
LDFAC	45	70	70	Load FAC
STFAC	35	35	35	Store FAC
AINT	135	220	310	Max when arg near 1.0
FIX	60	85	130	
INT	15	10	15	Plus FIX time and space
FLOAT	55	55	90	
FPADD	305	260	370	(add) Requires OR1
		270	490	(sub)
FPCMP	125	140	500	Comparison
FPMUL	140	450	560	Requires OR1
FPDIV	185	1070	1390	(DIV) Requires OR1
OR1	255			Entry/ Exit support

All functions and conversion routines require the complete basic operations package.

7.1.2 Transcendental Functions - FPAC

Name	Size	Exec. typ	Times max	Comments
FPATN	245	6400	6400	Requires FR1 & FR2
FPEXP	150	7500	7400	Requires FR1 & FR2
FPLN/LOG	280	6000	6500	Requires FR1 & FR2
FPSQRT	130	2850	2850	Requires FR1
FPCOS/SIN	110	5050	5050	Requires FR1, FR2, & FR3
FPTAN	110	8500	8500	Requires FR1, FR2, & FR3
FPXTOI	165	8300	8300	Requires FR1
FR1	30			Exception Routines
FR2	80			Polynomial Routines
FR3	110			Trig Support Routines

7.1.3 Conversion Routines – FPAC

Name	Size	Exec. Times	Comments
ASCBIN	290	1700 + n	n=920/digit (Req. CR1)
BINASC	600	1900 4500	Requires CR1
CR1	120		Conversion Support

7.2 8051 DPAC Routine Sizes and Execution Times

The sizes shown below are in bytes of program space. The execution time is given in CPU cycles for the 8051. Each CPU cycle is 12 oscillator periods.

7.2.1 Basic Operations Set – DPAC

Name	Size	Exec.	Times	Comments
		typ	max	
LDFACD	70	160	160	Load FAC
STFACD	50	110	110	Store FAC
DAINT	185	1000	1320	Max when arg near 1.0
DFIX	130	190	235	
DINT	25	5	20	Plus DFIX time and space
DFLOAT	120	165	255	Requires OR1
DPADD	350	750	1100	(add) Requires OR1
		850	1260	(sub)
DPCMP	170	600	1280	Comparison
DPMUL	180	1380	1530	Requires OR1
DPDIV	205	5300	6900	(DIV) Requires OR1
OR1	280			Entry/Exit Support

All functions and conversion routines require the complete basic operations package.

7.2.2 Transcedental Functions - DPAC

Name	Size	Exec.	Times	Comments
		typ	max	
DPATN	390	24000	24000	Requires FR1 & FR2
DPEXP	275	34900	34900	Requires FR1 & FR2
DPLN /LOG	345	23000	24400	Requires FR1 & FR2
DPSQRT	185	18500	18500	Requires FR1
DPCOS	210	23400	23400	Requires FR1, FR2, & FR3

/SIN				
DPTAN	140	28000	28000	Requires FR1, FR2, & FR3
DPXTOI	175	26000	26000	Requires FR1
FR1	35			Exception Routines
FR2	80			Polynomial Routines
FR3	130			Trig Support Routines

7.2.3 Conversion Routines – DPAC

Name	Size	Exec.	Times	Comments
DASCBN	375	7000 + n		n=2500/digit (Req. CR1)
DBNASC	720	7300	24000	Requires CR1
CR1	275			Power of ten routine

7.2.4 Precision Translation Routines – DPAC

Name	Size	Exec.	Times	Comments
		typ	max	
SINGLE	115	125	150	
DOUBLE	60	125	125	

8 Operation Summary

8.1 Single Precision Operations

Name	Operation	Description	Stack
LDFAC	Load	FAC = OPN	0
STFAC	Store	OPN = FAC	0
FPADD	Add	FAC = FAC + OPN	4
FPMUL	Multiply	FAC = FAC * OPN	4
FPDIV	Divide	FAC = FAC / OPN	4
FPRDIV	Divide	FAC = OPN / FAC	4
FPCMP	Compare	A per FAC - OPN	8
FLOAT	FAC = Float of Integer in B:A		0
INT	B: A = INT (OPN) {largest int <= fpn in OPN}		2
FIX	B: A = FIX (OPN) {integer part of fpn in OPN}		0
AINT	FAC = FLOAT of largest int <= fpn in OPN		0
FPATN	Arctangent	FAC = ATN (FAC)	12
FPCOS	Cosine	FAC = COS (FAC)	12
FPEXP	e to power	FAC = e**FAC	12
FPLN	Natural log	FAC = LN (FAC)	12
FPLOG	Common Log	FAC = LOG (FAC)	12
FPSIN	Sine	FAC = SIN (FAC)	12
FPSRT	Square Root	FAC = SQRT (FAC)	12
FPTAN	Tangent	FAC = TAN (FAC)	12
FPXTOI	fpn to power	FAC = FAC**i {B: A = power}	12
ASCBIN	FAC = fpn in ASCII pointed to by DPTR		10
BINASC	fpn in ASCII pointed to by DPTR = FAC		13

fpn	IEEE Floating Point Number
FAC	Floating Point Accumulator (fpn)
OPN	DPTR points to Operand (fpn)

8.2 Double Precision Operations

Name	Operation	Description	Stack
LDFACD	Load	FAC = OPN	0
STFACD	Store	OPN = FAC	0

DPADD	Add	FAC = FAC + OPN	4
DPMUL	Multiply	FAC = FAC * OPN	4
DPDIV	Divide	FAC = FAC / OPN	4
DPRDIV	Divide	FAC = OPN / FAC	4
DPCMP	Compare	A per FAC - OPN	8

DFLOAT	FAC = Float of Integer in R7: R6: B: A	2
DINT	R7: R6: B: A = largest int <= fpn in OPN	4
DFIX	R7: R6: B: A = integer part of fpn in OPN	2
DAINT	FAC = Float of largest int <= fpn in OPN	0

DPATN	Arctangent	FAC = ATN (FAC)		14
DPCOS	Cosine	FAC = COS (FAC)		14
DPEXP	e to power	FAC = e**FAC		14
DPLN	Natural log	FAC = LN (FAC)		14
DPLOG	Common Log	FAC = LOG (FAC)		14
DPSIN	Sine	FAC = SIN (FAC)		14
DPSRT	Square Root	FAC = SQRT (FAC)		14
DPTAN	Tangent	FAC = TAN (FAC)		14
DPXTOI	fpn to power	FAC = FAC**i	{B: A = power}	14

SINGLE	FAC = conversion of FAC to single precision	0
DOUBLE	FAC = conversion of FAC to double precision	0

DASCBN	FAC = fpn in ASCII pointed to by DPTR	14
DBNASC	fpn in ASCII pointed to by DPTR = FAC	16

fpn	IEEE Floating Point Number
FAC	Floating Point Accumulator (fpn)
OPN	DPTR points to Operand (fpn)

The following files comprise the FPAC/DPAC delivery package:

FILE	DESCRIPTION
readme.txt	Release notes
lib*.a51	All source code of GoFast library
TestAssistant*.*	Test Assistant project (VC++ 6.0). Test Assistant sends the test cases and receives the test results via serial port automatically.
test*.*	Accuracy Test source code (accuracy.c/h) and GoFast library header file gofast.h.
test\build\*.*	Silicon Laboratories IDE project accuracy.wsp, makefile of Microsoft NMAKE (accuracy.mak), and batch file accuracy.bat.
test\c8051f330\*.*	Board support code.

8.3 Accuracy Test

The Accuracy Test program outputs the result of each GoFast function with inputs selected for each. Because of the limitation of ROM and RAM size, only one function can be tested each time. Users should set the related macro to 1 in the file accuracy.h to select which to test.

The Test Assistant program is used to send the test cases to Accuracy Test program automatically via serial port. When the test is finished, the test results can be saved and compared with the correct results file, which is saved in directory result. See readme.txt for details.

The Accuracy Test source code may be used as an example of how the GoFast library routines are used by an application program. Execution allows the user to exercise the features of the GoFast library and to view its operation.

8.4 Performance Test

Performance Test outputs each function's execution time. The program can be used to test the GoFast library or Keil standard math library. Because of the limitation of ROM and RAM size, only one routine can be tested at a time. Users should set the related macro to 1 in the file bench.h.

The Test Assistant program is used to send the test cases to the Performance Test program automatically via serial port and calculate the time, which is sent by the Performance Test program via serial port. When the test is finished, the test results can be saved. See readme.txt for details.

Performance Test Results

(Tested on Silicon Laboratories C8051F330 board, 12MHz)

Function	**Single**		**Double**
	GoFast	**Keil**	**GoFast**
add	53.76	**20.252**	116.437
sub	59.064	**22.273**	115.7
mul	71.852	**21.486**	200.024
div	155.601	**89.242**	675.331
cmp	37.768	**8.956**	75.115
fabs	**3.25**	3.792	3.405
sin	566.35	**324.068**	2480.175
cos	557.513	**321.301**	2509.376
tan	962.696	**525.083**	3108.613
atan	753.152	**401.841**	2876.746
log	731.001	**396.874**	2626.336
log10	778.619	**416.921**	2787.991
exp	881.564	**515.099**	4179.886
pow	1639.917	**953.854**	6935.416
sqrt	361.673	**220.053**	2224.42
ceil	**15.242**	186.61	56.75
floor	**15.071**	186.739	56.616
char to float	**14.946**	20.669	36.133
unsigned char to float	**12.584**	20.669	34.976
short to float	15.309	**12.502**	34.043
unsigned short to float	15.059	**12.502**	33.708
long to float	13.1	**11.96**	30.894
unsigned long to float	9.75	**9.502**	29.133
float to char	**19.167**	20.69	28.401
float to unsigned char	**18.942**	20.69	28.175
float to short	21.175	**20.856**	31.258
float to unsigned short	**17.917**	20.856	30.929
float to long	27.588	**21.19**	43.171
float to unsigned long	27.588	**21.19**	43.173
abs for char	**2.74**	2.896	
abs for short	**2.573**	2.652	
abs for long	**3.238**	4.196	
double to single	49.711		
single to double	50.575		

9 Application Notes

9.1 On-Chip Data Memory (#01)

Problem: A number of customers have expressed a need to have the floating point operands manipulated by the libraries reside in on-chip data memory (OCDM).

The 8051 has 128 bytes of OCDM. Depending on the particular application, 8 to 32 bytes may be used for register banks, 15 to 28 bytes may be assigned to FPAC/DPAC processing, some amount of storage is allocated for stack space, with the remainder of the OCDM space available for user variables. If the FPAC/DPAC operands are to reside in OCDM, the user variable space must now hold not only the operands, but one to four floating point temporaries (4 to 32 bytes) needed by the conversion and function routines (if they are used). As a result, the OCDM fills up rapidly. For this reason, we do not encourage this technique if alternatives are available. (The release of successor 8051 hardware with additional OCDM will, however, make this approach practical).

Solution: From the programmer’s standpoint, the changes below to use OCDM operands instead of external data memory operands do not affect the method of addressing operands; that is, the operand address is still passed in DPTR (although a R/W memory operand in OCDM is only affected by the low byte, DPL). We deem the minor inefficiency of using a 16 bit addressing scheme for a seven bit address space acceptable to minimize code changes and preserve compatibility.

Old (Change to FPOPNS after label STFAC :)

```
****            MOV       R6, DPL
****            MOV       R7, DPH
```

New (Change to the following)

```
****            MOV       R1, DPL
```

Old (Change 4 occurrences of the following after STFAC:)

```
****            MOVX      @DPTR, A
```

New (Change those 4 occurrences to the following)

```
****            MOV       @R1, A
```

Old (Change 3 occurrences of the following after STFAC:)

```
****            INC       DPTR
```

New (Change those 3 occurrences to the following)

```
****            INC       R1
```

```
Old     (Delete the occurrence of the following after STFAC:)
****                    MOV         DPL, R6
****                    MOV         DPH, R7
Old     (Change after label FDATA)
****                    MOVX        A, @DPTR

New     (Change to the following)
****                    MOV         R1, DPL
****                    MOV         A, @R1

Old     (Change in module DPOPNS after label STFACD:)
****                    MOV         R6, DPL
****                    MOV         R7, DPH

New     (Change to the following)
****                    MOV         R1, DPL

Old     (Change 3 occurrences of the following after STFACD:)
****                    MOVX        @DPTR, A

New     (Change those 3 occurrences to the following)
****                    MOV         @R1, A

Old     (Change 2 occurrences of the following after STFACD:)
****                    INC         DPTR

New     (Change those 2 occurrences to the following)
****                    INC         R1

Old     (Delete the occurrence of the following after STFACD:)
****                    MOV         DPL, R6
****                    MOV         DPH, R7

Old     (Change 3 lines after the label DDATA:)
****                    MOVX        A, @DPTR

New     (Change to the following)
****                    MOV         R1, DPL
****                    MOV         A, @R1
```

9.2 On-Chip Memory Usage (#107)

Problem: Customers have expressed a need to be able to reduce the on-chip memory resources consumed by FPAC and DPAC routines.

Solution: Basic operations for the 8051 FPAC and DPAC require one byte of bit addressable, on-chip memory. The conversion routines and functions consume a second byte. If this resource is at an absolute premium, and exception conditions do not need to be signalled, it is possible to reduce the total number of bytes needed by FPAC/DPAC to one.

STEP 1 –Remove all references to unnecessary bits

These are found in the result routines FOPRSL, INFRSL, NANRSL, UNFRSL, ZERRSL, ONERSL (in functions), and the corresponding double precision routine. Delete all assembly statements that reference:

FACBIT	(Generally, ANL or ORL operations)
UNFFLG	(Generally, SETB operations)
NANFLG	(Generally, SETB operations)
INFFLG	(Generally, SETB operations)

STEP 2 –Change bit declarations

Delete the declarations for UNFFLG, NANFLG, and INFFLG. Change the remaining bit declarations to:

```
DPFLAG     EQU     FACBIT.4
SIGFLG     EQU     FACBIT.3
MANSGN     EQU     FACBIT.2
FNCSGN     EQU     FACBIT.1
FNCSEC     EQU     FACBIT.0
```

9.3 Reentrancy (#02)

Problem: A number of customers have expressed a need to have the library be reentrant. Because the FPAC/DPAC routines use fixed memory locations for the floating point accumulator and associated variables, they are not inherently reentrant.

Solution: To make the FPAC/DPAC routines reentrant, it is necessary to call a state preservations routine at the start of each interrupt service routine which uses FPAC/DPAC and to call a state restoration routine before returning from an interrupt service routine that invoked the preservation routine. These subroutines preserve static variables and register values on a stack.

The following data must be saved/restored across interrupt servicing:

1) Registers including ACC, B, PSW, DPTR, and some FPAC/DPAC bit registers.

2) The floating point accumulator (FAC) and associated resident variables.

3) Temporaries used by FPAC functions, if functions are (A) included and (B) used at an interrupt level.

There will need to be an area to store preserved variables. While this could be done on the hardware stack, the amount of data involved would consume a large portion of the on-chip data memory. The approach of choice would be to use a region of external data memory.

The routine that saves and restores the machine/FPAC state is not, and cannot be, reentrant. This means that any interrupt that would invoke the state save/restorations routine must be masked during certain critical periods.

Steps of state preservation:

A. Save registers on the hardware stack

```
        CLR     IE.7        ;Prevent other interrupts

        PUSH    PSW         ;Save machine registers
        PUSH    ACC
        PUSH    B
        PUSH    DPL
        PUSH    DPH
        PUSH    FACBIT      ;Save FPAC bit registers
        PUSH    CNVBIT

;;;  Change register bank number in PSW or
;;;  save registers R0-R7 here
```

B. Save FPAC variable area to external data memory

```
****        MOV     A,R4        ;Save R4,R6,R7 only if bank switch
****        PUSH    ACC
****        MOV     A,R6
****        PUSH    ACC
****        MOV     A,R7
****        PUSH    ACC
```

```
                MOV     DPL,XSPL   ; Get external mem stack pointer
                MOV     DPH, SXPH

                MOV     R6,#FACBAS  ;Base address
                MOV     R7,#27         ;(15 for FPAC, 27 for DPAC)
        XMPUSH  MOVX    A, @R6
                INC     R6
                MOVX    @DPTR,A
                INC     DPTR
                DJNZ    R7,XMPUSH

****            MOV     R7,#HIGH(EXTMEM)       ;Only if functions
****            MOV     R6,#LOW(EXTMEM)
****            MOV     R4,#4*8        ;4*4 if single precision routines
****            CALL    STCOPY         ;(See XPCNVT module)

                MOV     XSPL,DPL   ;Update external mem stack pointer
                MOV     XSPH, DPH
****            POP     ACC            ;Only if pushed previously
****            MOV     R7,A
****            POP     ACC
****            MOV     R6,A
****            POP     ACC
****            MOV     R4,A

****            SET     IE.7           ;Allow interrupts
```

Steps of state restorations:

A. Restore the FPAC variable area from external data memory

```
                CLR     IE.7           ;Disallow interrupts

                MOV     A,XSPL         ;Get external mem stack pointer
                CLR     C
                SUBB    A,#27+4*8  ;Size of the FPAC region
                MOV     XSPL,A         ;Update ext mem sp
                MOV     R6,A           ;***DPL if no functions
                MOV     A,XSPH
                SUBB    A,#00
                MOV     R7,A           ;***DPH if no functions

****            MOV     DPTR,#EXTMEM   ;Only if functions
****            MOV     R4,#4*8        ;Size of FPAC function temp area
****            CALL STCOPY
```

```
****              MOV     DPL,R6
****              MOV     DPH,R7

                  MOV     R6,#FACBAS   ;Base address
                  MOV     R7,#27       ;(15 for FPAC, 27 for DPAC)
        XMPOP     MOVX    A,@DPTR
                  INC     DPTR
                  MOVX    @R6,A
                  INC     R6
                  DJNZ    R7,XMPOP
```

B. Restore Registers from the hardware stack

**** Restore R0-R7 if not banked switched

```
                  POP     CNVBIT       ;Restore FPAC bit registers
                  POP     FACBIT

                  POP     DPH          ;Restore machine state
                  POP     DPL
                  POP     B
                  POP     ACC
                  POP     PSW

****              SET     IE.7         ;reenable interrupt
```

Special Notes:
This code assumes XSPH:XSPL is intialized to point to the lowest address byte of an external memory block that provides (at most) 59*maximum reentrant level bytes for storage. Note that 10 bytes are used on the hardware stack per reentrant level.

9.4 The Floating Point Accumulator Structure in FPACs (#152)

FPACs which are built around the structure of a FAC (Floating Point Accumulator) present the user with what may be called a single address virtual machine. An understanding of how single address architectures work may give the user more insight into effective utilization of FPAC/FACs.

Probably the most durable computer architecture is based on a register-memory organization. Registers are in many ways an address space separate from the memory. Bulk data storage is the purpose of the memory area while operations can only be done on data within registers. This separation permits and requires machine instructions to transfer data between the two regions. Once data has been moved into register(s), a different class of instructions is used to manipulate it.

The FAC organization of FPACs attempts to mimic this structure. A set of the target processor’s resources (generally on-chip data memory) is dedicated for the Floating Point Accumulator. Whereas hardware – implemented registers can only be accessed by the instructions of the architecture – and thus may be protected from invalid references – the FAC resources are reserved only by a software convention. Thus, the FAC area is “off-limits” to conventional access and must only be dealt with by the FPAC routines supplied.

An audit of the facilities of an FPAC/FAC look very much like a simple single address machine. A set of transfer routines are used to move data between the target processor’s bulk memory and the FAC pseudo-register. Operations are performed with the value in the FAC and, perhaps, a value in bulk memory.

One key difference must exist between a hardware implementation of single address machine and a software implementation: address specification. Hardware implemented architectures almost invariably include one or more “address fields” in their instruction format. Since software implemented architectures usually cannot directly use this hardware format, a different approach to address specification will normally be necessary. The FPAC/FAC routines take the approach of using a particular register in the underlying target processor to hold the memory address of the operand to be involved in an operation. Thus, while single address machines may use just one instruction to perform an operation, FPAC/FACs will normally require two instructions: one to place the operand’s address into the appropriate register followed by a subroutine call to perform the operation. (When sequential operations use the same memory address, however, reloading the address register is unnecessary since the FPAC/FAC routines almost always preserve the incoming value of the address register.)

Knowing the software conventions used in implementing the FAC, it becomes clear how to make an FPAC/FAC “re-entrant”. Hardware-based re-entrancy comes from the ability to save the entirety of the state of the machine when a context switch (such as an interrupt) occurs (and, of course, the ability to restore the state when switching back!). Since the hardware has no idea that a FAC is in use (and that it’s an extension to the state information that needs to be saved and restored), the user must – if FPAC re-entrancy is needed – manually save and restore the set of target processor resources that comprise the FAC. This will include a variety of temporary storage areas and flags in addition to what might be considered just the FAC proper. (The FPAC routines which use a memory operand in combination with a value in the FAC first transfer the memory operand to temporary registers in preparation for later work). This can be a lengthy process, especially for double precision FPACs, and since the need to have FPAC re-entrancy on target processors that have FPAC/FACs is typically slight or non-existent, the steps to make an FPAC/FAC re-entrant are rarely taken.

Currently available FPACs which use a FAC organization include: 8051 FPAC/DPAC, 68HC11 FPAC/DPAC, 6301 FPAC/DPAC, 6801 FPAC/DPAC, 8096 FPAC/DPAC, Z-80 FPAC/DPAC, and 8085 FPAC/DPAC.

9.5 General Overview of Accuracy and Precision (#122)

For effective use of FPAC/DPAC routines, it is important to have an understanding of the meaning of accuracy and precision in the representation of values in the IEEE single and double precision format. Although the examples below refer to single precision representation, the difficulties shown are present, though to a lesser extent, with double precision representation.

Single precision FPAC has a mantissa precision of 24 bits, which is approximately 7.2 decimal digits. The “approximately” qualifier belies the fact that conversion between decimal and binary representations can be inexact. The ASCII to binary conversion routine chooses the binary value closest to the decimal value argument, in the event an exact decimal-binary conversion is not possible.

The table below illustrates this inexact conversion problem. The first column gives a decimal value. The second column contains the decimal value of the IEEE single precision number closest to the decimal value.

Decimal Value	Decimal Value (Closest Single Precision Value)
1.0	1.0
1.1	1.10000002384
1.01	1.00999999046
1.001	1.00100004673
1.0001	1.00010001659
0.1	1.0000000149e-1
0.01	9.9999997765e-2
0.001	1.0000000475e-3
0.0001	9.9999997474e-4

The representational difficulty becomes apparent when computing the difference of two “close” numbers. For example, the operation:

1.001-1.0 = 0.001

After conversion of IEEE single precision representation becomes:

1.00100004673-1.0 = 1.00004673e-3

In like manner,

0.001 + 1.0 – 1.0

will yield 1.00004673e-3 instead of the expected 0.001 (or 0.0010000000475) because of the representation characteristics.

These problems become particularly apparent when using the EXP and LN functions. In the case of the LN function (and the corresponding log), the algorithm effectively subtracts 1.0 from the argument. As shown in the example above, this subtraction can magnify imprecisions. Thus, an unexpected value can result from apparently accurate value, if the user is unaware of the underlying representation aspects of FPAC/DPAC.

9.6 Tailoring Double Precision Function Accuracy (#148)

The IEEE floating point standard defines a single precision and a double precision format. The single precision format has about seven decimal digit accuracy while the double precision format has nearly sixteen decimal digit accuracy. Quite often, more than single precision accuracy may be needed for a particular application, but accuracy offered by the double precision format far exceeds the requirement. The result of unnecessary accuracy may be excessive computation time.

In general, it is not practical to modify DPAC operation routines in an attempt to trade lesser accuracy for faster execution time. However, certain functions in the double precision function library are amenable to simple accuracy reductions that will speed the routines. These functions are the exponentiation (DPEXP) and the sine and cosine (DPSIN and DPCOS).

These routines are suitable for accuracy/speed tradeoffs because after a relatively brief range reduction or scaling sequence, these functions use somewhat lengthy polynomial approximations. The approximation polynomials are such that by shrinking their degree, accuracy will be lost but speed will be gained. Other functions use a split-domain approach to keep the approximation polynomial small (DPATN, DPLN/DPLOG), use an iterative algorithm not suitable for early termination (DPSQRT) , or have a fixed computation sequence that needs to remain unchanged (DPXTOI, DPTAN).

The exponentiation routine has a fifteen (or, in more recent DPACs, a thirteen) degree polynomial. Roughly speaking, each degree equates to one decimal digit of accuracy. Hence, to go from sixteen digit accuracy to ten digit accuracy, a reduction of six degrees could be realized.

To implement DPEXP accuracy reduction, two modifications to the source code in the DPFNCS module need to be made. First, the label associated with the start of the DPEXP constants table should be moved to the appropriate value. In the example given, to reduce the polynomial degree by six, the label at the start of the constant table should be associated with the seventh constant in the list rather than the first constant. (Note that by moving the label instead of deleting the constants, it’s easier to back out of the change). The label name depends on the processor the DPAC was written for:

Processor	DPEXP Constant Table Label
8051, 8085, Z-80	DEXCNS
80386, 68HC11, 6301, 6801	DEXCON
68000, 8096	DEXPCN
8086	EXPCONS

In addition, the symbolic name indicating the number of constants in the table needs to reflect the correct number of constants in the abbreviated table. Again in the example given, the table length constant in DPEXP would be changed from its delivery value of sixteen to ten. The symbolic name is, as before, dependent on the processor:

Processor	DPEXP Table Length Name
8051, 8085, Z-80	DNEXCN
80386, 68HC11, 6301, 6801	DNEXCN
68000, 8096	NDEXPC
8086	NEXPCN

To estimate the reduction in computation time, take the typical function time as delivered then subtract the result of multiplying the number of constants removed from the table by the sum of the typical multiply time plus the typical addition time.

The trigonometric routines gain about two decimal digits of accuracy for each constant in their polynomial table. Hence, continuing with the example reduction to ten digits of accuracy, three constants could be removed from the approximation polynomial tables for sine and cosine.

Processor	Sine		Cosine	
	Table	Length	Table	Length
8051, 8085, Z-80	DSINCN	DNSNCN	DCOSCN	DNCOCN
80386, 68HC11, 6301, 6801	DSICON	DNSICN	DCOCON	DNCOCN
68000, 8096	DSINCN	NDSINC	DCOSCN	NDCOSC
8086	SINCON	NSINCON	COSCON	NCOSCON

A word of warning when doing this. The approximation error with the delivered functions is generally randomly distributed within less than two bits of actual function value. By truncating the polynomials, the approximation error becomes regular and monotonically increasing across sections of function domains. If the required function accuracy is met even at the points of maximum error, then this will not matter. If, however, the algorithm using reduced accuracy functions is sensitive to error distribution, either added accuracy will be required or different approximation polynomials with better error distributions will be needed.

9.7 Implementing an XˆY Operation

The GoFast library provides an operation to compute X to the I^{th} power (I is an integer). As shown in the FPAC/DPAC manual, all combinations of operands yield a deterministic result of the "proper" type. If at all possible, this routine should be used.

Expanding to the general case of a floating point number taken to a floating point power is somewhat more complicated because a floating point power can have a "special" value (+INF, -INF, or NaN), or it can be a non-integral value applied to a negative number (for example, -1.5 ˆ 0.5).

The traditional means for implementing an X to the Y^{th} power is to follow this sequence of steps:

LN(X)	Take the natural logarithm of X
Y*LN(X)	Compute the natural logarithm of the result
EXP(Y*LN(X))	Take e to the power of the product

One difficulty in implementation occurs when the first step, taking the natural logarithm of X, "fails" – that is, the LN function returns NaN, indicating an invalid operation, and the X operand was not NaN. This will happen if X is a negative number or if X is –INF. Another failure will occur when 1.0 is taken to a +INF or –INF. The second step of the algorithm given above must, in this instance, multiply 0.0 by an INF value. In this particular case, we know the result should be 0.0 (because 1.0 to any power is 1.0), but the multiply routine, as per the IEEE proposed standard, returns NaN. A corrolary to this second instance is taking +INF to the 0th power.

If the particular application that needs an XˆY operation can insure that the X value is a positive number and that the Y value is a number, then the algorithm given will work well. Should an application not have limited domains on the operands, the implementer will need to pre-screen the operands to handle cases where the algorithm does not function. The first step in doing this is completing (and, possibly, changing) the table below so that the XˆY operation returns a meaningful value for all relevant combinations of operands.

Result Range for XˆY operation

	Y					
X	-INF	Y < 0	0	0 < Y	+INF	NaN
-INF	0*	0*	1	?	?	NaN
X < -1	?	?	1	?	?	NaN
-1	?	?	1	?	?	NaN
-1 < X < 0	?	?	1	?	?	NaN
0	+INF	+INF	NaN	0	0	NaN
0 < X < 1	+INF	****	1	***	0*	NaN
1	1	1	1	1	1	NaN

1 < X	0*	***	1	****	+INF	NaN
+INF	0*	0*	1	+INF	+INF	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

*: Underflow
**: result could be 0 (an underflow) or a number (less than 1.0)
***: result could be a number (greater than 1.0) or +INF

The implementer can use the XTOI function as a template for doing the pre-screening operation.

9.8 Error Codes and FPAC Conversion Routines (#123)

The FPAC application note details the behavior of the ASCII-to-binary and binary-to-ASCII conversion utilities with respect to setting of the error code and sticky bits (error flags). In general, it applies to all the FPAC libraries, although some of the specifics of routine operations will vary (for example, in an instance discussed below, when and if an ASCII-to-binary conversion routine will experience overflow during the processing of a value too small to represent is library dependent).

9.8.1 ASCBIN and DASCBN – ASCII to binary conversion routines

The general flow of these conversion routines is to compute a mantissa value then scale the mantissa value by multiplying it by a computed power of ten. In both cases, the computations are done in the floating point domain, using the FPAC addition, multiplication, and division routines of the appropriate precision. Because the normal routines are used, the error code is set and, under certain circumstances, sticky bits may be set. (Note: certain custom versions of FPAC perform the mantissa and scaling operations in the integer domain – for extended precision at the cost of added code space and execution time. Although operations are not performed by floating point routines the range of the result is checked, so these custom libraries follow the same error code and error flag conventions.)

If the result of the conversion is within the range of the precision in use, the error code returned will be 0 (no error), and no sticky bits will be set by the conversion routine. If the value returned is smaller than is representable, a zero value will be returned, the error code will be set to 1 (for underflow), the sticky bit for underflow (generally called UNFFLG) will be set, and in certain cases the overflow flag (generally called INFFLG) may be set (scaling to a negative power of ten is done with division by the appropriate positive power of ten – which may be too big to represent). If the value returned is larger than is representable, an infinity value of the appropriate sign will be returned, the error code will be set to 2 (for overflow), and the sticky bit for overflow (generally called INFFLG) will be set. If the ASCII data provided is syntactically invalid, (no mantissa digits, for example), a Not-a-Number representation (NaN) will be returned, the error code will be set to 3 (for invalid operation), and the sticky bit for invalid operations (generally called NANFLG) will be set.

9.8.2 BINASC and DBNASC – binary to ASCII conversion routines

The general flow of these conversion routines to divert special values for independent processing, otherwise to scale the incoming value by a computed power of ten to put it in a scientific notation form. A string processing routine is then used to convert a selected range of values to a floating point ASCII representation.

The special values: zero, signed infinity, a Not-a-Number (NaN), are "converted" to ASCII by what amounts to a string copy. Since no floating point routines are used, conversion of these special values does not change the error code nor set any sticky bits.

The computing of the scaling power of ten, and the scaling of the values not selected for special processing is done in the floating point domain, using the FPAC addition, multiplication, and division routines of the appropriate precision. Because the normal FPAC routines are used, and because all of the floating point operations performed to convert a standard value to ASCII yield a standard value, the error code will be set to 0 (no error) and no sticky bits will be set. (Note: certain custom versions of FPAC perform these operations in the integer domain – for extended precision at the cost of added code space and execution time. Since integer domain routines are used, the error code and sticky bits are left unchanged.)

If it is desirable to have the binary-to-ASCII conversion routine set the error code to the appropriate value (and/or to set the appropriate sticky bit), some code should be added to the segments that process the special case values. In general, this will mean replacing a return instruction with a register load instruction (to the desired error code) and a jump to the error code setting routine (in the xPOPNS module). Of course, the specific register and routine to jump to will vary with the particular library involved. The details can be found by observing the steps taken by the special value return routines in the xPOPNS module; they are invoked by the basic operations routines and the range checking routine to return a special value and to set the error code and sticky bits.

9.9 Understanding and Using FPAC Routine Timing Estimates (#135)

The routine by routine timing estimates provided in FPAC manuals can aid in estimating the performance of programs using FPAC modules. Unfortunately, as with many compilation of statistics, improper use of the estimates can produce misleading results. The purpose of this application note is to assist in the correct utilization and interpretation of the timing numbers.

Timing Units

The timing estimates are given in units of a particular processor's instruction cycles. Depending on the particular processor, the oscillator or clock frequency is often a multiple of the instruction cycle frequency (in most cases, the factor is two or three although in one instance it is twelve!). Included as part of the FPAC data sheet is a table giving typical instruction cycle periods. Usually, manufacturer data sheets refer to instruction periods or "processor" clock frequency. The tabulation of instruction execution times given in processor specifications is normally proceeded or followed by a discussion of clock frequency and instruction cycle timing.

Estimate Values

The basic FPAC routines have two timing estimate values. The *typical* value is an estimate of the number of instruction cycles that a broad range of expected values, or value pairs, will consume, on average. Of course, execution times for specific values will differ from the given typical value.

The *maximum* execution cycle count is, as the name states, the longest processing time that a routine can consume. In general, this is a very rare occurrence.

Estimate Conditions

The system conditions assumed for the timing estimates are important factors in the timing estimates. A direct conversion from instruction cycle count to execution time, (multiplying the cycle count by an appropriate instruction period), is often made. If the system does not have a constant clock frequency, (the system may use processor clock stretching to "hide" refresh or DMA cycles), some form of compensation in the form of an effective clock frequence is necessary to meaningfully apply the FPAC timing estimates.

Another important assumption made about system behavior is zero-wait state memory for data and code reference. While the 8051 and 6809 require this, most processors have some facility for inserting wait states in memory access operations. Clearly, the increased memory access time associated with wait state memory will add execution time to FPAC routines. Manufacturers data sheets usually contain some guidelines for estimating the performance reduction caused by various memory speeds.

On processors that can overlap instruction execution with memory cycles, FPAC routines are implemented to be as insensitive to longer memory accesses times as practical. This can be seen in some cases by unusual instruction ordering, memory accesses that may sometimes unnecessarily pre-fetch data items to avoid additional memory cycles, and maximum practical use of register-resident values. While these steps reduce the potential performance reduction, inevitably, the use of wait state memory will slow FPAC routine execution. System hardware designers, utilizing manufacturer's projections, should be able to estimate a fairly accurate performance reduction factor.

Finally, in the case of processors with dynamic bus sizing, FPAC assumes that all memory references are made with the maximum bus width supported by the processor.

Typical-only Timing Estimates
The FPAC functions do not have "maximum" timing values. This reflects the extreme difficulty in choosing values that will consume the maximum instruction cycles. Instead, the function timing estimates are built up from a combination of basic operator timings – tuned for the manner in which the basic operators are used with the particular function. Once these basic operator timings, added to the execution time of surrounding instructions, are summed, an adjustment is applied to insure that the typical timing given will only rarely be exceeding in practice. The adjustment factor is routine dependent and is generally in the range of three to ten percent.

9.10 The Polynomial Function Evaluation Routine (#139)

The functions supplied by the FPAC/DPAC library are of general utility. Some users, though, need special purpose functions in addition to the standard FPAC/DPAC functions. When constructing the code to perform the special purpose function evaluation, it can be necessary to compute the value of a polynomial function. In addition, a general polynomial function evaluator may be required for some FPAC/DPAC applications. Making the polynomial evaluator, which is internal to the FPAC/DPAC library, can serve both needs.

Evaluation of any polynomial function requires three items: the function argument (the "x" value), the list of coefficients, and the number of coefficients. The table below illustrates the method that is used to supply the function argument.

FPAC/DPAC	Argument	Use Routine
Z-80	FAC	LDPAC or LDFACD
8085	FAC	LDPAC or LDFACD
8051	FAC	LDPAC or LDFACD
8096	FAC	LDPAC or LDFACD
68HC11	FAC	LDPAC or LDFACD
6801	FAC	LDPAC or LDFACD
6301	FAC	LDPAC or LDFACD
8086	user mem	– none – (DS:DI pointer)
80386	registers	LDOP1 or DLDOP1
68000	registers	GETFP1 or GETDP1

The 68000 routine is internal to FPAC/DPAC. The user must place the argument value on the stack, then a four byte value (simulating a return address), then call the internal routine to properly load the appropriate registers. For more details, see the header comments associated with the internal routines.

The coefficient list is pointed to by a register containing the base address of a list of floating point values in IEEE floating point format. The first constant in the list is applied to the highest power of the function argument, followed by succeedingly lower power argument coefficients, finally reaching the constant (zero power) coefficient. The table below indicates the register used to hold the coefficient table pointer.

FPAC/DPAC	Coefficient Pointer Register	
Z-80	HL	
8085	HL	
8051	DPTR	coefficients reside in ROM
8096	FPPNTR	coefficients reside in ROM
68HC11	X	
6801	X	
6301	X	
8086	SI	coefficients reside in CS
80386	EBX	coefficients reside in CS
68000	A1	

The number of coefficients, (which is the degree of the polynomial plus one), is supplied in a register. The specific register is given below:

FPAC/DPAC	Number of Coefficients	
Z-80	A	
8085	A	
8051	A	
8096	FACTMP	(on-chip dedicated RAM location)
68HC11	A	
6801	A	
6301	A	
8086	AX	
80386	ECX	
68000	D7	

Once the values have been prepared for the polynomial evaluator, the user calls one of two routines. The two routines provided allow a polynomial in powers of the function argument or in squared powers of the function argument (generally associated with trigonometric functions). The table below gives the names of these routines.

FPAC/DPAC	Standard	Squared
Z-80	XSER or DXSER	XXSER or DXXSER
8085	XSER or DXSER	XXSER or DXXSER
8051	XSER or DXSER	XXSER or DXXSER
8096	XSER or DXSER	XXSER or DXXSER
68HC11	XSER or DXSER	X2SER or DX2SER
6801	XSER or DXSER	X2SER or DX2SER
6301	XSER or DXSER	X2SER or DX2SER
8086	XSER	XSQRSER
80386	XSER or DXSER	X2SER or DX2SER
68000	XSER or DXSER	X2SER or DX2SER

Note that the single and double precision forms of the polynomial evaluation routines for the 8086 are in different modules. If they are made public, they would have to have different names.

The polynomial function evaluators return the result in the same way the parameter is supplied. The "inverse" routine to the loading routine is used to store the result for the 8051, 8085, Z-80, 8096, and 80386 FPAC/DPACs. The 8086 FPAC/DPAC overwrites the argument at DS:DI with the result.

The 68000 FPAC/DPAC has internal routines FOPRSL and DOPRSL which round, compress, and place the in-register value on the stack. To invoke one of these routines, the user places the return address in A0, and then jumps to the appropriate routine's entry point.

As with many efforts in software implementation, it is often easier to learn from existing functional code than to independently develop and debug code. For this reason, it is advisable to examine the exponentiation (EXP) and the simple trigonometric routines (SIN/COS) for examples of how to use the standard and squared power polynomial evaluation routines. These functions can serve as effective templates for the successful implementation of custom FPAC/DPAC functions.

10 References

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

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